Quadia/Duet User's Manual

Quadia/Duet User's Manual

The Quadia/Duet User's Manual was prepared by the technical staff of Innovative Integration on February 5, 2009.

For further assistance contact:

Innovative Integration
2390-A Ward Ave
Simi Valley, California 93065

PH: (805) 578-4260
FAX: (805) 578-4225

email: techsprt@innovative-dsp.com
Website: www.innovative-dsp.com

This document is copyright 2009 by Innovative Integration. All rights are reserved.

VSS \ Distributions \ Quadia \ Documentation \ Manual \ QuadiaMaster.odm

#XXXXXX

Rev 1.0

Table of Contents

Chapter 1. Introduction....13

Real Time Solutions!....13
Vocabulary....13
What is Velocia? 14
What is Quadia/Duet?....15
What is Malibu? 15
What is C++ Builder?....15
What is Microsoft MSVC?....15
What kinds of applications are possible with Innovative Integration hardware?....15
Why do I need to use Malibu with my Baseboard?....15
Finding detailed information on Malibu....16
Online Help....16
Innovative Integration Technical Support....16
Innovative Integration Web Site....17
Typographic Conventions....17

Chapter 2. Windows Installation....18

Host Hardware Requirements....18
Software Installation....18
Starting the Installation 19
The Installer Program....20
Tools Registration....22
Bus Master Memory Reservation Applet....22
Hardware Installation....23
After Power-up....24

Chapter 3. JTAG Hardware Installation....25

JTAG Emulator Hardware Installation (for DSP boards Only)....25
PCI Pod-Based Emulator Installation....25
Baseboard Installation....26
A Few Considerations BEFORE Power-up....26
It cannot be overemphasized: 26
After Power-up....27
Code Composer Studio Setup with II Jtag....27
Setting up for a single processor with Spectrum Digital USB Jtag....31
Setting up for Multi Processors with Spectrum Digital USB Jtag....34
Borland Builder Setup and Use....37
Automatic saving of project files and forms during debugging....37
Static-binding of built executables. 38
Appropriate library and include paths. 39

Chapter 4. DSP Baseboard Overview....41

The Velocia Baseboard Family....41
The Baseboard Device Driver....43
Multiple Baseboards....44
Slave Accesses....44
Data Transfers....45

Message Packet I/O....46
Class Libraries....46
Malibu....46

Chapter 5. A Tour of Malibu....48

Class Groups In Malibu....48
Utility Classes....49
Events and Event Handlers....49
Windows Synchronization....50
Thread Classes....50
Multi-threading Utility Classes....50
Buffer and Message Classes....50
Hardware and Hardware Support Classes....50
Baseboards and PMC Modules....51
Subsystem Interfaces....51
Target I/O Streaming Classes....52
Analysis Classes....53
Vector (1D) Signal Processing Components....53
Data Storage and Retrieval....53
Conversion Functions....54
Using the Malibu Component Suite....54
Creating a Streaming Application in Visual C++....54
Creating the Malibu Objects....54
Initializing Object Properties and Events....56
Event Handler Code....57
Loading COFF Files....59
Loading Logic Files....59

Chapter 6. About the Baseboard....60

Velocia Family Overview....60
Quadia/Duet Overview....60
Processing Cluster....62
Connectivity....63
PCI Buses....63
Data Plane....64
Global Memory Pool....65
Timing and Synchronization Features....65
The Pismo Class Library....66
Simple To Use....66
Not Just for C++ Experts....67
Unique Feature Support for each Baseboard....67
Digital Signal Processor....67
DSP External Memory....67
DSP Initialization....67
DSP JTAG Debugger Support....70
FPGA JTAG Support....70
Using the Malibu Baseboard Components....72
PCI Interrupt Configuration and Compatibility....73
DSP Programming on the Baseboard....73
Device Drivers....74
Advantages of using DSP/BIOS drivers....74

How to use a DSP/BIOS driver....74
Driver-specific control functions....76
Driver Buffer Model....76
Driver Types....77
Driver Implementation....77
DMA-enabled Drivers....78
Simplified Use....78
Multitasking Friendly....79
Analog Timebase Objects....79
Timebase Usage....79
Interrupt Handling....80
Interrupts in a C++ Environment....80
The Pismo Solution....80
Class Irq....81
Interrupt Lock Classes....81
Interrupt Binder Templates....82
Class InterruptHandler....82
Class ClassMemberHandler Template....82
Class FunctionHandler Template....83
EDMA and QDMA Handling....83
Class DmaSettings....84
Class Qdma....84
Class Edma....85
Linked and Chained blocks....87
Class EdmaMaster....88
Quadia and Duet Example Programs....88
The Next Step: Developing Custom Code....89

Chapter 7. Host/Target Communications....90

Overview....90
CPU Busmastering Interface....91
CPU Busmastering Implementation....91
Packet Based Transfers....91
Blocking Interface....91
Maximum Transfer Size....91
Malibu Library Host Support for CPU Busmastering....92
Packet Notification Events....92
Target (Pismo Library) Support for CPU Busmastering....92
Packetized Message Interface....93
Message Mailbox Emulation....93
The Message System....93
Host-side Message Objects....94
Target Side Message Objects....95
Message Communication....96
C++ Terminal I/O....97
Target Software....97
Tutorial....97

Chapter 8. Building a Target DSP Project....99

Writing a Program....104
Host Tools for Target Application Development....104

Components of Target Code (.cpp, .tcf, .cmd, .pjt)....104
Edit-Compile-Test Cycle using Code Composer Studio....105
Automatic projectfile creation....105
Rebuilding a Project....105
IIMain replaces main....105
Running the Target Executable....105
Note:....106
Anatomy of a Target Program....106
Use of Library Code....107
Example Programs....107
The Next Step: Developing Custom Code....108

Chapter 9. Developing Host Applications....109

Borland Turbo C++....109
Other considerations:....110
Microsoft Visual Studio 2005....111
DialogBlocks....113
Summary....113

Chapter 10. Applets....114

Common Applets....114
Registration Utility (NewUser.exe)....114
Reserve Memory Applet (ReserveMemDsp.exe)....115
Data Analysis Applets....115
Binary File Viewer Utility (BinView.exe)....115
Target Programming Applets....116
Target Project Copy Utility (CopyCcsProject.exe)....116
Demangle Utility (Demangle.exe)....116
COFF Section Dump Utility (CoffDump.exe)....116
JTAG Diagnostic Utility (JtagDiag.exe)....117
RtdxTerminal - Terminal Emulator....117
Important Note:....117
Terminal Emulator Menu Commands....117
The File Menu....118
The DSP Menu....118
The Form Menu....119
The Help Menu....119
Options Tab:....119
Display Group....120
Sounds Group....121
Coff Load Group....121
Debugger Group....121
Terminal Emulator Command Line Switches....122
Applets for the C64x DSP Processor....123
COFF Downloader (Download.exe)....123
ConfigRom: C64x DSP EEProm Configuration Utility (C64xEeprom.exe)....123
Applets for the Quadia/Duet Baseboard....124
Baseboard Finder (Finder.exe)....124
Target Number....124
Blink....124
On/OFF....124

PCI Logic Update Utility (Eeprom.exe)....124
Logic Download Utility (LogicLoader.exe)....124

Chapter 11. Target Peripheral Devices....126

Quadia Memory Map....126
Quadia/Duet Control Registers....126
PCI Control Register....127
PCI Status Register....128
Cluster 0 FPGA Control Register....128
Cluster 1 FPGA Control Register....129
DSP Control Registers....129
Cluster FPGA Status Registers....129
DSP Memory Map....130
Card Controls....130
Reset Control....130
Host Reset....130
DSP Resets....131
PLL Resets....131
FPGA Resets....131
DSP Interrupts....131
DSP INT4 and DSP INT5 Source Assignment....132
Card Synchronization Features....133
Clock Generation....133
Synchronizing to external clocks....133
Clock and Trigger Sharing....133
Communications....133
System Level Data Communications Design....134
PCI Bus....135
Using PCI for Command and Control....135
Using PCI for Data Communications....136
PCI Interrupt Mapping....136
PMC Interrupt Control....136
StarFabric (Quadia Rev C Only)....137
Data Plane....137
DSP FIFOLinks....137
DSP to DSP Data Plane Component....139
DSP FIFOLink FIFO Data....140
DSP FIFOLink FIFO Status Registers....140
DSP FIFOLink FIFO Control Registers....141
DSP FIFOLink Reset Control Register....141
SFP Links....142
What are SFP Modules....142
SFP Link Component....143
SFP Data Port....143
SFP Status Register....143
SFP FIFO Status Register....143
SFP Cabling....144
SFP Error Rates....144
DSP and FPGA Communications....144
Using DSP EMIF B....144
Controlling Data Flow to the DSPs....145

IO Devices....145
Clock Generation PLLs....145
PLL Device....145
PLL Reference Input....145
PLL Frequency Generation....146
PLL Connections....147
PLL Control Registers....147
PLL Data Registers....149
PLL Use....149
PMC/XMC Modules....149
Note : XMC support has been added to Quadia beginning with Rev E. 149
PMC Mechanicals....149
PMC PCI Interface 150
PMC J4 Support....150
XMC Support 150
XMC Rocket IO Pairs....150
XMC Controls....151
Custom XMC Applications....152
Memory Pool....152
Memory Pool Architecture....152
Memory Pool Performance....153
Memory Pool Use Rules....153
Software Support....154
FPGAs....154
Velocia FPGA....155
Reprogramming the Velocia FPGA....155
Cluster FPGA Devices....155
Cluster FPGA Connections....156
FPGA DSP Connections....156
Cluster FPGA Memory....156
Cluster FPGA Miscellaneous Connections....157
Cluster FPGA Power Supplies....157
Loading the Cluster FPGA image....157
External IO Input to Cluster FPGAs....159
Rear Terminal IO (Revisions D and above)....159
Duet PXI Support....159

Chapter 12. Connector Pinouts and Physical Information....161

Quadia Connectors....161
PMC Private IO Connector (JN4)....161
PMC IO Connector (JN8)....162
JP3 – FPGA JTAG Connector....164
JP14 – FPGA JTAG Connector....165
JP1 – DSP JTAG....166
....167
JP10 – Power Input Connector (Test Only)....167
JE1, JE2 – Logic Testpoint Connectors (Quadia Rev C only)....168
....170
External IO J1/J2 170
PCI-X Enable JP15....171
Sync Connector JP2 (Rev C only)....171

JP11 – Factory Power Test Connector....172
JP7, JP8 – SFP Connectors....173
JN1, JN5 – PMC Connectors....175
JN2, JN6 – PMC Connectors....177
JN3, JN7 – PMC Connectors....180
JP4 – JTAG Test Connector for PCI Bridges and Miscellaneous....183
JP5 – Velocia V2Pro FPGA test points Connector....184
CJ3 – StarFabric and Ethernet Backplane Connector....184
P15, P16 – XMC Connectors (Quadia Rev E + only)....188
CJ5 - Compact PCI Rear Terminal User IO (Quadia Rev E + only)....190
Board Layouts....194
Duet Connectors....198
PMC Private IO Connector (JN4)....198
JP2 – FPGA JTAG Connector....199
JP6 – Velocia FLASH JTAG Connector....199
JP4 – Factory Power Test Connector....200
JP1 – DSP JTAG....201
JN1 – PMC Connector....202
JN2 – PMC Connectors....205
JN3 – PMC Connector....207
P1 – XMC Connectors....210
CJ2 - Compact PCI and PXI....212
....216

Chapter 13. Troubleshooting....218

Initialization Problems....218
The system does not recognize my board(s)....218
I created an EXE file and when I try to run it, the system requires a DLL which I don't have....218
What DLLs do I have to deploy with my newly created executable?....219
How do I know what DLLs my executable is dependent on?....219
DSP Hardware Problems....220
The I/O seems like it is not connected or doesn't work....220
How can I tell what version of logic I am using?....220
Quadia Hardware Problems....220
How do I update the logic?....220
I updated the logic, but it did not work....220

List of Tables

Table 1. Quixote 'C6416 DSP EMIF Control Register Initialization Values....68
Table 2. Quadia 'C6416 DSP EMIF Control Register Initialization Values....69
Table 3. Velocia Baseboard Components....72
Table 4. Velocia Family Baseboard Logic Configuration Methods....72
Table 5. Velocia Family Baseboard COFF Loading Methods....72
Table 6. Timebase Operations....80
Table 7. Interrupt Lock Classes....82
Table 8. Quadia/Duet Example Programs....88
Table 9. TIIMessage Header Field Access....94
Table 10. TIIMessage Data Section Interface....95
Table 11. IIMessage Header Field Access....95
Table 12. IIMessage Data Section Interface....96
Table 13. Message Sending and Receiving Methods....96
Table 14. Pismo Example Programs....108
Table 15. Quadia/Duet Baseboard PCI Memory Map (BAR 1)....127
Table 16. Quadia/Duet Baseboard Control Register (Write, BAR1 +0x0)....127
Table 17. Quadia/Duet Baseboard Status Register (Read, BAR1 + 0x0)....128
Table 18. Cluster 0 FPGA Control Register (write, BAR1 + 0x18)....128
Table 19. Cluster 1 FPGA Control Register (write, BAR1 + 0x20)....129
Table 20. DSP Control Registers (write, DSP0 = +0x44, DSP1 = +0x48, DSP2 = +0x4C, DSP3 = +0x50)....129
Table 21. Cluster FPGA Status Registers (read, Cluster0 = +0x1C, Cluster1 = +0x24)....129
Table 22. DSP EMIF A Memory Map....130
Table 23. DSP Interrupt Assignments....132
Table 24. DSP INT4/5 Source Selection Register....132
Table 25. Maximum Data Rates....135
Table 26. PCI Interrupt Assignments....136
Table 27. DSP FIFO Mesh Connections....139
Table 28. FIFOLink Status Registers....141
Table 29. DSP FIFOLink FIFO Control Registers....141
Table 30. DSP FIFOLink Reset Control Register....141
Table 31. A Sample of Compatible SFP Modules....142
Table 32. SFP Status Registers....143
Table 33. SFP FIFO Status Register....144
Table 34. PLL Clock Connections to Cluster FPGAs....147
Table 35. PLL Control Registers (write, PLL0 = +0x34, PLL1 = +0x35)....148
Table 36. PLL Data Registers (write, PLL0 = +0x28, PLL1 = +0x2C)....149
Table 37. XMC Rocket IO Pairs....151
Table 38. XMC Control and Support Signals....152
Table 39. Summary of Compatible Cluster FPGA Devices....155
Table 40. FPGA Power Supplies....157
Table 41. Quadia JTAG Chains and Connectors....158
Table 42. Duet JTAG Chains and Connectors....158
Table 43. Duet PXI Signals....159
Table 44. PMC JN4....161
Table 45. PMC JN8....162
Table 46. FPGA JTAG Connector Pinouts....164
Table 47. JTAG Connector for Velocia FPGA and FLASH....165

Table 48. JP1 DSP JTAG Connector Pinouts....166
Table 49. JP10 Power Input Connector (Test Only)....168
Table 50. FPGA testpoint Connector....169
Table 51. JN1, JN5 – PMC Connectors....175
Table 52. PMC JN4 198
Table 53. FPGA JTAG Connector Pinouts....199
Table 54. JTAG Connector for Velocia FLASH....200
Table 55. JP1 DSP JTAG Connector Pinouts....201
Table 56. JN1 – PMC Connectors....202
Table 57. Windows driver files....218

List of Figures

Figure 1. Vista Verificaton Dialog....19
Figure 2. Innovative Install Program....20
Figure 3. Progress is shown for each section....21
Figure 4. ToolSet registration form....22
Figure 5. BusMaster configuration....22
Figure 6. Installation complete....23
Figure 7. Quadia Block Diagram....42
Figure 8. Duet Block Diagram....43
Figure 9. Bus-mastering efficiently transfers data between target and host memory....45
Figure 10. Bus mastering transfers are always initiated by the target DSP or PMC....46
Figure 11. Quadia Block Diagram....60
Figure 12. Duet Block Diagram....61
Figure 13. Quadia Processing Cluster Block Diagram....62
Figure 14. Quadia PCI Architecture....63
Figure 15. Quadia Data Plane Connections....64
Figure 16. Quadia DSP JTAG Chain....70
Figure 17. Quadia FPGA JTAG Chain (Rev F and above)....71
Figure 18. Quadia FPGA JTAG Chain (Rev A-E)....71
Figure 19. Duet FPGA JTAG Chain....71
Figure 20. Messaging System Objects....94
Figure 21. RtdxTerminal Options....120
Figure 22. Data Plane Connections on Quadia....137
Figure 23. Quadia DSP FIFOLink Mesh....138
Figure 24. Quadia Data Plane FIFO Mesh....139
Figure 25. Rocket IO Link Component....140
Figure 26. Some SFP modules (Picture courtesy of MRV Communications)....142
Figure 27. PLL Reference Crystal Specifications....146
Figure 28. Simplified View of Memory Pool Controller....153
Figure 29. Simplified View of Cluster FPGA Connections....156
Figure 30. PMC JN4 Connector Pin out....162
Figure 31. PMC JN8Connector Pin out....163
Figure 32. Power Connector Pin Positions (side view, from front of connector, showing connector keying and locking tab along with printed circuit board position)....168
Figure 33. Quadia Rev C Board Layout....194
Figure 34. Quadia Rev D/E Board Layout....195
Figure 35. Quadia Rev F Board Layout....196
Figure 36. Quadia Rev G Board Layout....197
Figure 37. PMC JN4 Connector Pin out....198
Figure 38. Duet Rev B Board Layout....217

Chapter 1. *Introduction*

Real Time Solutions!

Thank you for choosing Innovative Integration, we appreciate your business! Since 1988, Innovative Integration has grown to become one of the world's leading suppliers of DSP and data acquisition solutions. Innovative offers a product portfolio unrivaled in its depth and its range of performance and I/O capabilities .

Whether you are seeking a simple DSP development platform or a complex, multiprocessor, multichannel data acquisition system, Innovative Integration has the solution. To enhance your productivity, our hardware products are supported by comprehensive software libraries and device drivers providing optimal performance and maximum portability.

Innovative Integration's products employ the latest digital signal processor technology thereby providing you the competitive edge so critical in today's global markets. Using our powerful data acquisition and DSP products allows you to incorporate leading-edge technology into your system without the risk normally associated with advanced product development. Your efforts are channeled into the area you know best ... your application.

Vocabulary

What is Velocia?

Velocia is an advanced architecture DSP baseboard that integrates a high performance Texas Instruments TMS320C64xx DSP and Xilinx high density programmable logic with high performance peripherals such as PMC modules, analog IO and interconnectivity interfaces. The powerful combination of the DSP and FPGA provide signal processing speed and flexibility for almost any DSP application. Each baseboard features a PCI backbone connecting the DSP, PMC, peripherals and StarFabric interfaces. The StarFabric interface (PICMG 2.17) provides unlimited and extremely flexible interconnection to other DSP cards, IO cards and host processor systems. Each Velocia card incorporates a high performance IO system with either on-board peripherals like A/D and D/As, or one or more PMC sites accommodating a wide range of I/O options.

What is Quadia/Duet?
Quadia and Duet are Velocia family DSP baseboards. Quadia features four DSPs and dual FPGAs, whereas Duet has two DSPs and a single FPGA. Both baseboards are designed for computation intensive applications such as wireless base stations and test equipment, RADAR, image processing and co-processing. Quadia has four Texas Instruments TMS320C6416 DSPs (two for Duet) and two VP40 standard, Xilinx Virtex2 Pro FPGAs (one for Duet). Processor and system connectivity is both flexible and real-time by virtue of the PCI bus and data plane architecture of the card. Dual PMC sites as well as other peripherals provide modular IO expansion to suit a variety of applications.

What is Malibu?
Malibu is the Innovative Integration-authored component suite, which combines with the Borland, Microsoft or GNU C++ compilers and IDEs to support programming of Innovative hardware products under Windows and Linux. Malibu supports both high-speed data streaming plus asynchronous mailbox communications between the DSP and the Host PC, plus a wealth of Host functions to visualize and post-process data received from or to be sent to the target DSP.

What is C++ Builder?
C++ Builder is a general-purpose code-authoring environment suitable for development of Windows applications of any type. Armada extends the Builder IDE through the addition of functional blocks (VCL components) specifically tailored to perform real-time data streaming functions.

What is Microsoft MSVC?
MSVC is a general-purpose code-authoring environment suitable for development of Windows applications of any type. Armada extends the MSVC IDE through the addition of dynamically created MSVC-compatible C++ classes specifically tailored to perform real-time data streaming functions.

What kinds of applications are possible with Innovative Integration hardware?
Data acquisition, data logging, stimulus-response and signal processing jobs are easily solved with Innovative Integration baseboards using the Malibu software. There are a wide selection of peripheral devices available in the Matador DSP product family, for all types of signals from DC to RF frequency applications, video or audio processing. Additionally, multiple Innovative Integration baseboards can be used for a large channel or mixed requirement systems and data acquisition cards from Innovative can be integrated with Innovative's other DSP or data acquisition baseboards for high-performance signal processing.

Why do I need to use Malibu with my Baseboard?
One of the biggest issues in using the personal computer for data collection, control, and communications applications is the relatively poor real-time performance associated with the system. Despite the high computational power of the PC, it cannot reliably respond to real-time events at rates much faster than a few hundred hertz. The PC is really best at processing data,

not collecting it. In fact, most modern operating systems like Windows are simply not focused on real-time performance, but rather on ease of use and convenience. Word processing and spreadsheets are simply not high-performance real-time tasks.

The solution to this problem is to provide specialized hardware assistance responsible solely for real- time tasks. Much the same as a dedicated video subsystem is required for adequate display performance, dedicated hardware for real-time data collection and signal processing is needed. This is precisely the focus of our baseboards – a high performance, state-of-the-art, dedicated digital signal processor coupled with real-time data I/O capable of flowing data via a 64-bit PCI bus interface.

The hardware is really only half the story. The other half is the Malibu software tool set which uses state of the art software techniques to bring our baseboards to life in the Windows environment. These software tools allow you to create applications for your baseboard that encompass the whole job - from high speed data acquisition, to the user interface.

Finding detailed information on Malibu
Information on Malibu is available in a variety of forms:

- Data Sheet (http://www.innovative-dsp.com/products/malibu.htm)
- On-line Help
- Innovative Integration Technical Support
- Innovative Integration Web Site (www.innovative-dsp.com)

Online Help

Help for Malibu is provided in a single file, Malibu.chm which is installed in the Innovative\Documentation folder during the default installation. It provides detailed information about the components contained in Malibu - their Properties, Methods, Events, and usage examples. An equivalent version of this help file in HTML help format is also available online at http://www.innovative-dsp.com/support/onlinehelp/Malibu.

Innovative Integration Technical Support

Innovative includes a variety of technical support facilities as part of the Malibu toolset. Telephone hotline supported is available via

Hotline (805) 578-4260 8:00AM-5:00 PM PST.

Alternately, you may e-mail your technical questions at any time to:

techsprt@innovative-dsp.com.

Also, feel free to register and browse our product forums at http://forum.iidsp.com/, which are an excellent source of FAQs and information submitted by Innovative employees and customers.

Innovative Integration Web Site

Additional information on Innovative Integration hardware and the Malibu Toolset is available via the Innovative Integration website at www.innovative-dsp.com

Typographic Conventions

This manual uses the typefaces described below to indicate special text.

Chapter 2. *Windows Installation*

This chapter describes the software and hardware installation procedure for the Windows platform (WindowsXP and Vista).

> **Do *NOT*** install the hardware card into your system at this time. This will follow the software installation.

Host Hardware Requirements

The software development tools require an IBM or 100% compatible Pentium IV - class or higher machine for proper operation. An Intel-brand processor CPU is *strongly recommended*, since AMD and other "clone" processors are not guaranteed to be compatible with the Intel MMX and SIMD instruction-set extensions which the Armada and Malibu Host libraries utilize extensively to improve processing performance within a number of its components. The host system must have at least 128 Mbytes of memory (256MB recommended), 100 Mbytes available hard disk space, and a DVD-ROM drive. Windows2000 or WindowsXP (referred to herein simply as *Windows*) is required to run the developer's package software, and are the target operating systems for which host software development is supported.

Software Installation

The development package installation program will guide you through the installation process.

> **Note:** Before installing the host development libraries (VCL components or MFC classes), you must have Microsoft MSVC Studio (version 9 or later) and/or Codegear RAD Studio C++ (version 11) installed on your system, depending on which of these IDEs you plan to use for Host development. If you are planning on using these environments, it is imperative that they are tested and known-operational before proceeding with the library installation. If these items are not installed prior to running the Innovative Integration install, the installation program will not permit installation of the associated development libraries. However, drivers and DLLs may be installed to facilitate field deployment.

You must have **Administrator Privileges** to install and run the software/hardware onto your system, refer to the Windows documentation for details on how to get these privileges.

Starting the Installation

To begin the installation, start Windows. Shut down all running programs and disable anti-virus software. Insert the installation **DVD**. If Autostart is enabled on your system, the install program will launch. If the DVD does not Autostart, click on Start | Run... Enter the path to the **Setup.bat** program located at the root of your DVD-ROM drive (i.e. **E:\Setup.bat**) and click "OK" to launch the setup program.

> **SETUP.BAT** detects if the OS is 64-bit or 32-bit and runs the appropriate installation for each environment. It is important that this script be run to launch an install.

When installing on a Vista OS, the dialog below may pop up. In each case, select "Install this driver software anyway" to continue.

Windows Security
Windows can't verify the publisher of this driver software
Don't install this driver software
You should check your manufacturer's website for updated driver software for your device.
Install this driver software anyway
Only install driver software obtained from your manufacturer's website or disc. Unsigned software from other sources may harm your computer or steal information.
See details

Figure 1. Vista Verificaton Dialog

The Installer Program

After launching Setup, you will be presented with the following screen.

Figure 2. Innovative Install Program

Using this interface, specify which product to install, and where on your system to install it.

1) Select the appropriate product from the Product Menu.

2) Specify the path where the development package files are to be installed. You may type a path or click "**Change**" to browse for, or create, a directory. If left unchanged, the install will use the default location of "C:\Innovative".

3) Typically, most users will perform a "Full Install" by leaving all items in the "**Components to Install**" box checked. If you do not wish to install a particular item, simply uncheck it. The Installer will alert you and automatically uncheck any item that requires a development environment that is not detected on your system.

4) Click the Install button to begin the installation.

> **Note:** The default "Product Filter" setting for the installer interface is "Current Only" as indicated by the combo box located at the top right of the screen. If the install that you require does not appear in the "Product Selection Box" (1), Change the "Product Filter" to "Current plus Legacy".

Each item of the checklist in the screen shown above, has a sub-install associated with it and will open a sub-install screen if checked. For example, the first sub-install for “Quadia - Applets, Examples, Docs, and Pismo libraries” is shown below.

The installation will display a progress window, similar to the one shown below, for each item checked.

Figure 3. Progress is shown for each section.

Tools Registration

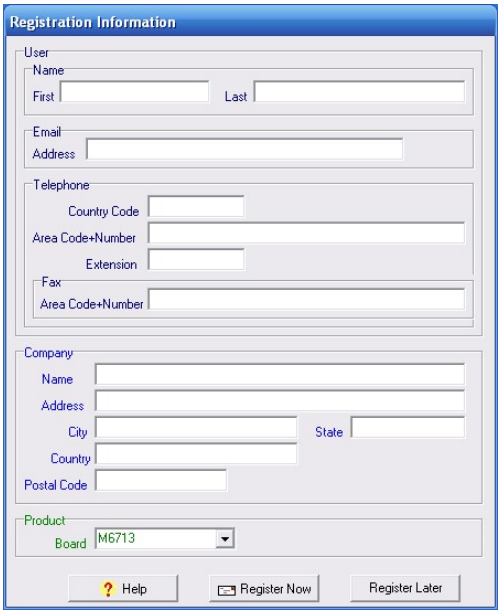


Figure 4. ToolSet registration form

At the end of the installation process you will be prompted to register. If you decide that you would like to register at a later time, click "Register Later".

When you are ready to register, click Start | All Programs | Innovative | <Board Name> | Applets. Open the New User folder and launch **NewUser.exe** to start the registration application. The registration form to the left will be displayed.

Before beginning DSP and Host software development, you must register your installation with Innovative Integration. Technical support will not be provided until registration is successfully completed. Additionally, some development applets will not operate until unlocked with a passcode provided during the registration process.

It is recommend that you completely fill out this form and return it to Innovative Integration, via email or fax. Upon receipt, Innovative Integration will provide access codes to enable technical support and unrestricted access to applets.

Bus Master Memory Reservation Applet.

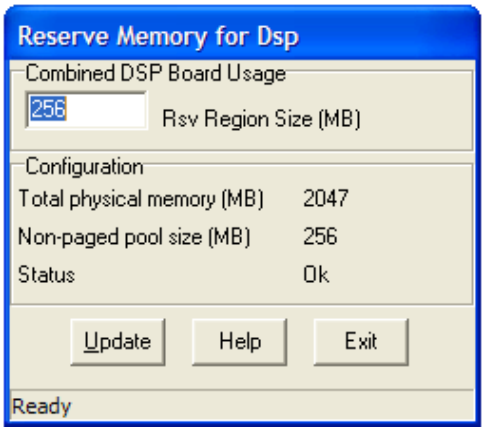


Figure 5. BusMaster configuration

At the conclusion of the installation process, **ReserveMem.exe** will run (except for SBC products). This will allow you to set the memory size needed for the busmastering to occur properly. This applet may be run from the start menu later if you need to change the parameters.

For optimum performance each Matador Family Baseboard requires 2 MB of memory to be reserved for its use. To reserve this memory, the registry must be updated using the ReserveMem applet. Simply select the **Number of Baseboards** you have on your system, click **Update** and the applet will update the registry for you. If at any time you change the number of boards in your system, then you must invoke this applet found in Start | All Programs | Innovative | <target board> | Applets | Reserve Memory.

After updating the system exit the applet by clicking the **exit** button to resume the installation process.

At the end of the install process, the following screen will appear.

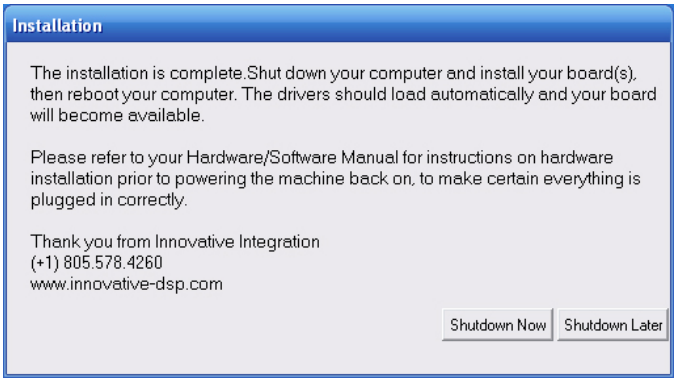

Installation
The installation is complete.Shut down your computer and install your board(s), then reboot your computer. The drivers should load automatically and your board will become available.
Please refer to your Hardware/Software Manual for instructions on hardware installation prior to powering the machine back on, to make certain everything is plugged in correctly.
Thank you from Innovative Integration
(+1) 805.578.4260
www.innovative-dsp.com
Shutdown Now
Shutdown Later

Figure 6. Installation complete

Click the "Shutdown Now" button to shut down your computer. Once the shutdown process is complete unplug the system power cord from the power outlet and proceed to the next section, "Hardware Installation."

Hardware Installation

Now that the software components of the Development Package have been installed the next step is to configure and install your hardware. Detailed instructions on board installation are given in the Hardware Installation chapter, following this chapter.

IMPORTANT: Many of our high speed cards, especially the PMC and XMC Families, require forced air from a fan on the board for cooling. Operating the board without proper airflow may lead to improper functioning, poor results, and even permanent physical damage to the board. These boards also have temperature monitoring features to check the operating temperature. The board may also be designed to intentionally fail on over-temperature to avoid permanent damage. See the specific hardware information for airflow requirements.

After Power-up

After completing the installation, boot your system into Windows.

Innovative Integration boards are plug and play compliant, allowing Windows to detect them and auto-configure at start-up. Under rare circumstances, Windows will fail to auto-install the device-drivers for the JTAG and baseboards. If this happens, please refer to the “TroubleShooting” section.

Chapter 3. *JTAG Hardware Installation*

JTAG Emulator Hardware Installation (for DSP boards Only)

First, the emulator hardware must be configured and installed into your PC. The emulator hardware is described in the table below:

Type	Features
Pod-based	Uses a special ribbon cable with integrated line drivers to connect the target DSP emulation signals to the JTAG debugger card. Usable on 3.3 volt or 5 volt designs. (Including 'C54x and 'C6x)

PCI Pod-Based Emulator Installation

To install the PCI pod based emulator, follow the instructions below:

5) Perform the board installation in an "ESD" or static safe workstation employing a static-dissipative bench mat. Wear a properly grounded wrist strap or other personal anti static device. Stand on an anti static mat or a static-dissipative surface.

6) Shut down Windows, power-off the host system and unplug the power cord.

7) Touch the chassis of the host computer system to dissipate any static charge.

8) Remove the card from its protective static-safe shipping container, being careful to handle the card only by the edges.

9) Touch the chassis of the PC to dissipate any built up static charge.

10) Securely install the JTAG board in an available PCI slot in the host computer.

11) Connect the JTAG pod to the host-pod cable. Connect the host-pod cable to the connector located on the end bracket of the JTAG PCI plug-in board.

Baseboard Installation

To install the baseboard:

12) Perform the board installation in an “ESD” or static safe workstation employing a static-dissipative bench mat. Wear a properly grounded wrist strap or other personal anti static device. Stand on an anti static mat or a static-dissipative surface.

13) Shut down Windows and power-off the host system and unplug the power cord

14) Touch the chassis of the host computer system to dissipate any static charge.

15) Remove the card from its protective static-safe shipping container, being careful to handle the card only by the edges.

16) Touch the chassis of the PC to dissipate any built up static charge.

17) Connect the 14-pin connector on the JTAG PCI pod to the DSP board JTAG connector (Non-DSP board users skip this step).

18) Securely install the baseboard into an available PCI slot in the host computer.

IMPORTANT: Many of our high speed cards, especially the PMC and XMC Families, require forced air from a fan on the board for cooling. Operating the board without proper airflow may lead to improper functioning, poor results, and even permanent physical damage to the board. These boards also have temperature monitoring features to check the operating temperature. The board may also be designed to intentionally fail on over-temperature to avoid permanent damage. See the specific hardware information for airflow requirements.

A Few Considerations BEFORE Power-up

Double-check all connections before applying power. Ensure that the JTAG and baseboard cards seated correctly in the slot.

It cannot be overemphasized:
Double check your cabling **BEFORE** connecting to the baseboard. **DO NOT** hot plug the cables. Hot plugging cables can cause latch-up of components on the card and damage them irreparably. Be aware that the cables to analog inputs are an important part of keeping signals clean and noise-free. Shielded cables and differential inputs (where applicable) help to control and reduce noise.

After Power-up
After completing the installation, boot your system into Windows.

Innovative Integration boards are plug and play compliant, allowing Windows to detect them and auto-configure at start-up. Under rare circumstances, Windows will fail to auto-install the device-drivers for the JTAG and baseboards. If this happens, please refer to the "TroubleShooting" section.

The next section is NOT used with the Non-DSP products. If the board you are installing is a Non-DSP product, the installation is complete.

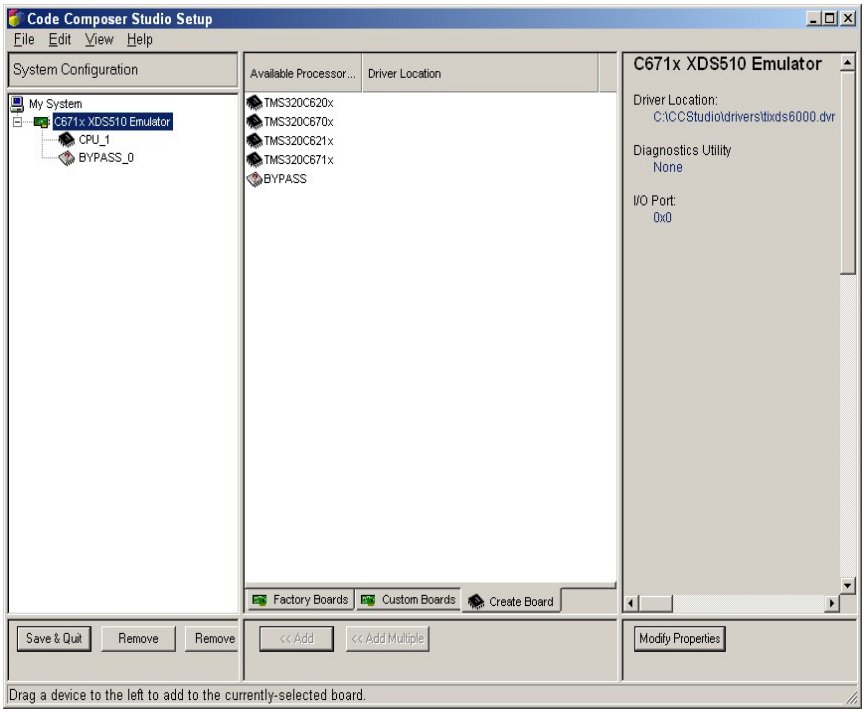
Code Composer Studio Setup with II Jtag

To setup Code Composer Studio and activate the Innovative Integration-supplied CodeHammer JTAG board driver, the Code Composer Studio Setup Utility must be run. Since the Code Hammer debugger is XDS510- compatible, Code Composer Studio setup must be configured to use the XDS510 driver for the C6xxx. To do this:

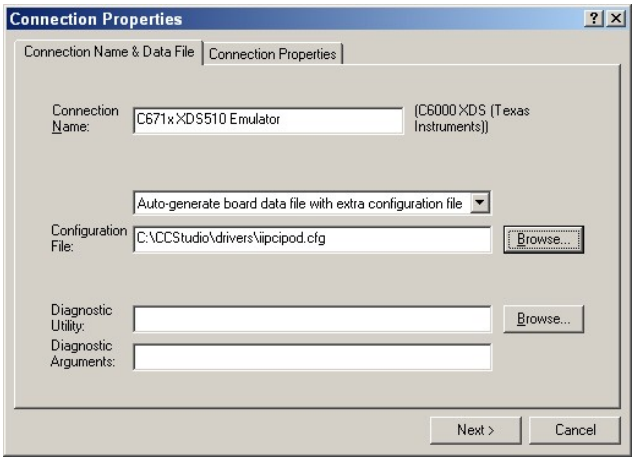
19) Launch the Code Composer Studio Setup Utility and remove the default simulator driver from the System Configuration. (right click the default simulator in the "System Configuration" pane and select "Remove")

20) Click the C6xxx XDS driver from the "Available Emulator Types" control within the setup utility and drag it into the "System Configuration" control.

21) Once your emulator is added, a list of Available Processors is presented. Add the appropriate processors for your board as shown in the example. The example shows a set-up that is configured for an SBC6713e baseboard. The C671x emulator is selected as the baseboard uses the TMS320C6713 DSP.

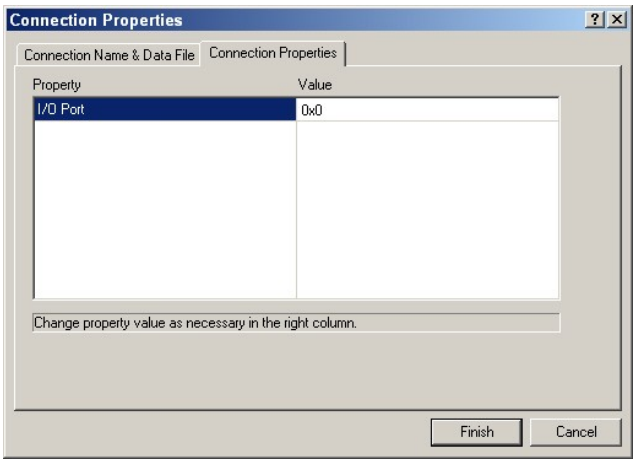


22) Right-click on the C6xxx XDS emulator in the "System Configuration" Pane and select "Properties…" to invoke the **Connection Properties** Dialog for the driver.



23) Under the "Connection Name & Data File" tab, the "**Connection Name**" edit box should match the emulator selected in the "**System Configuration**" Pane of the previous window. Change the "**Configuration File**" combo box to Auto-generate board data file with extra configuration file. Change the "**Configuration File**" edit box to "<drive>:\Cstudio\Drivers\IIPciPod.cfg". *<drive> is the letter for the drive onto which CCS is installed.*

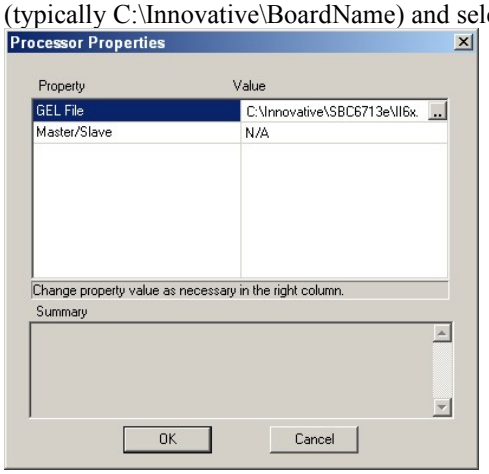
24) Click the "**Connection Properties**" tab. Set the I/O port value for the driver to virtual device address "0x0" and click "**Finish**".



25) The main Code Composer Studio Setup window is now back in focus. The processor must now be configured. To do this: select the processor as shown in the "**System Configuration**" Pane (in our example "CPU_1"). Right click "CPU_1" and select "Properties…".

26) The "**Processor Properties**" screen will be presented. Click GEL File, click the ellipsis (…) and navigate to the Innovative Integration board install directory

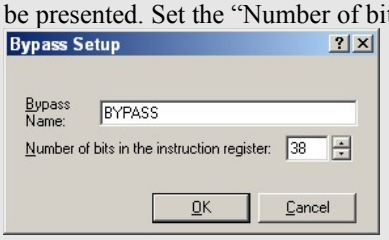
(typically C:\Innovative\BoardName) and select “II6x.gel”. Click “OK”.



27) Click “**Save & Quit**” to save the configuration and exit the setup tool. You will then be prompted to launch **Code Composer Studio**.

Note: For multi-target boards such as the Quadia, one processor should be added for each device in the JTAG scan path.

Note: The SBC6713e has (2) DSPs, a C6713 and a DM642. Typically the DM642 should be set to “BYPASS” by selecting “BYPASS” from the “Available Emulator Types” control within the setup utility and drag it into the “System Configuration” control. Once this is done, the following screen will be presented. Set the “Number of bits in the instruction register” to “38” and click “OK”



If you encounter difficulty launching CCS

28) Run the **JtagDiag.exe** utility (Start | All Programs | Innovative | Common Applets | JTAG Diagnostics) to reset the debugger interface.

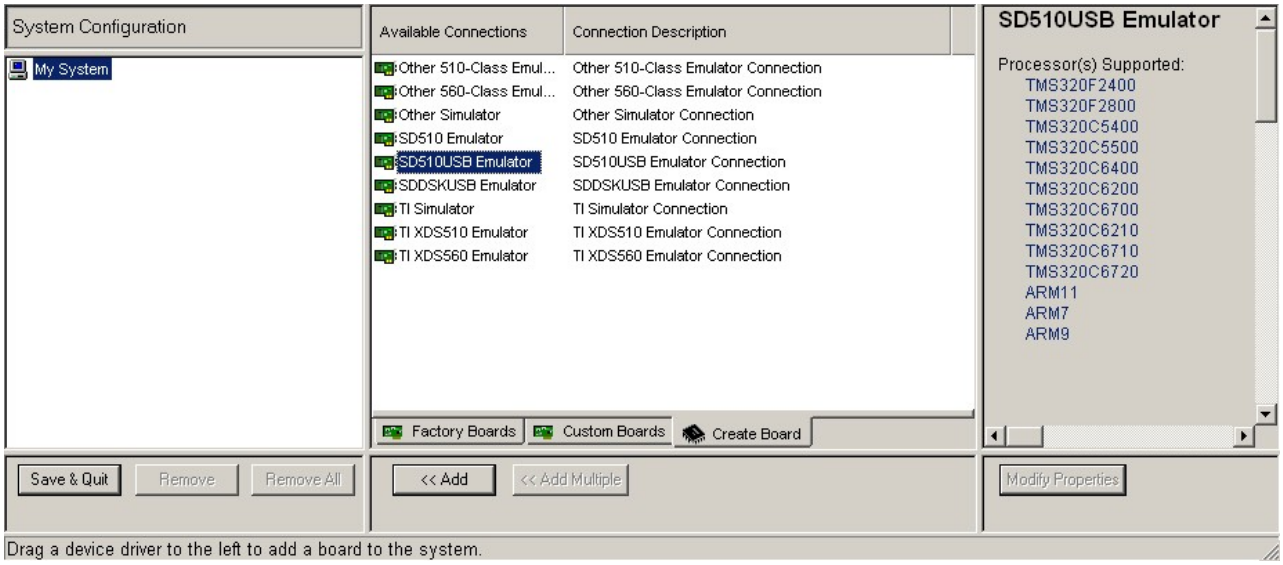
29) Run the board **Downloader utility** (Start | All Programs | Innovative | <Board Name> |<Applets> — Open the **Downloader Folder** and double click “**Downloader.exe**”) and press the Boot button (*Light Bulb icon*), to boot a default target application.

30) Restart Code Composer Studio.

Setting up for a single processor with Spectrum Digital USB Jtag.

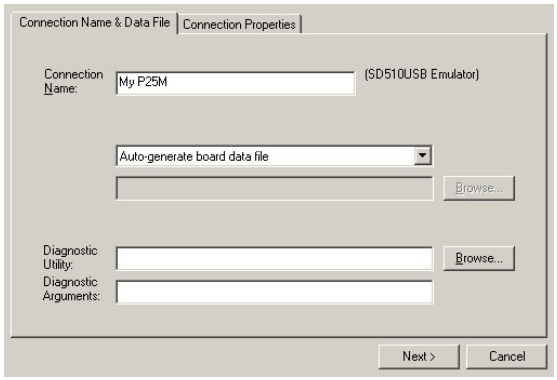
First remove any previous setups in the CCS Setup application.

Add one of the USB SD type driver.



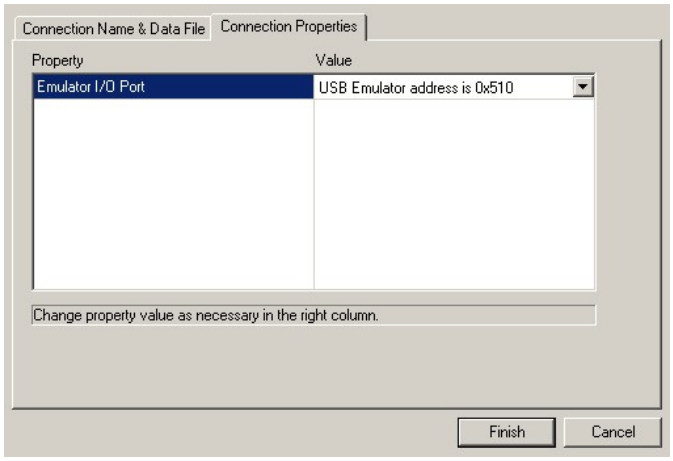
You will see the following screen.

Fill out the name of the board you are using, this can be any name you like.



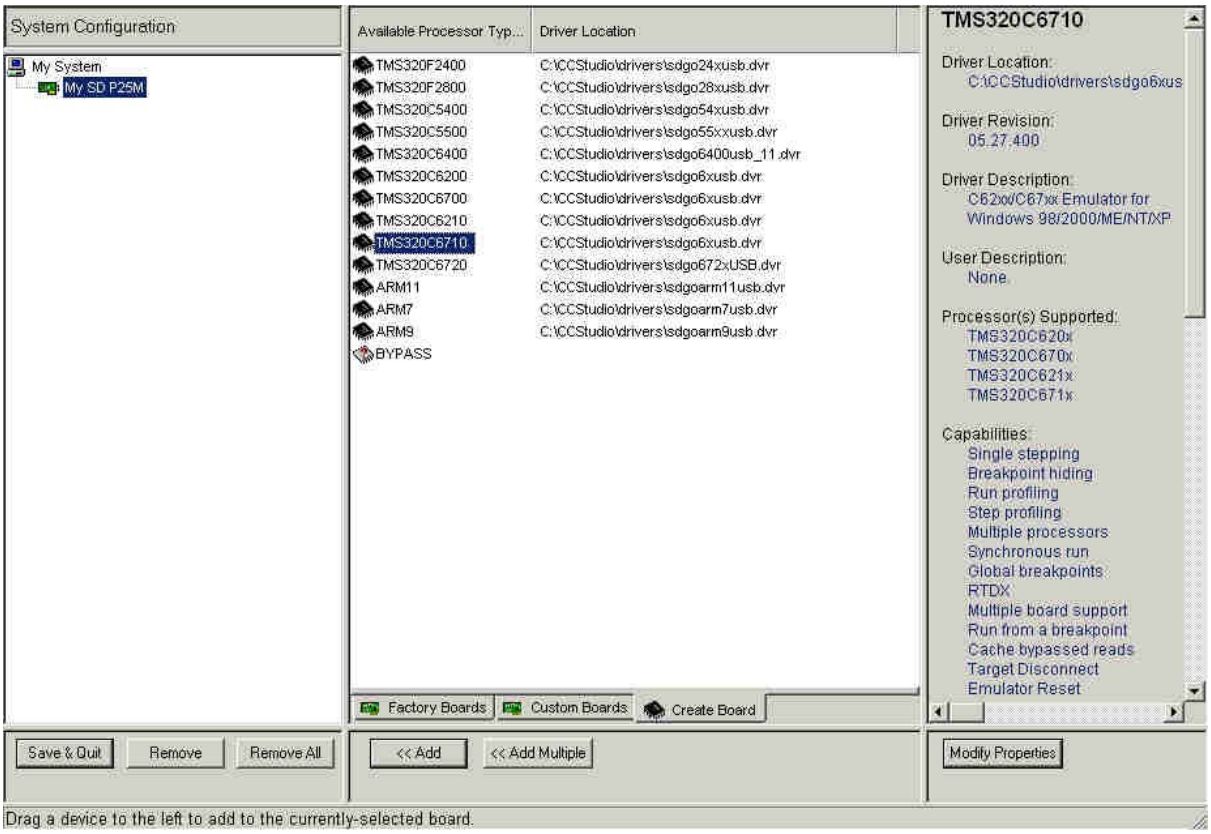
Hit next or move to the next tab

This address should match up with the address in the SdConfig.exe utility

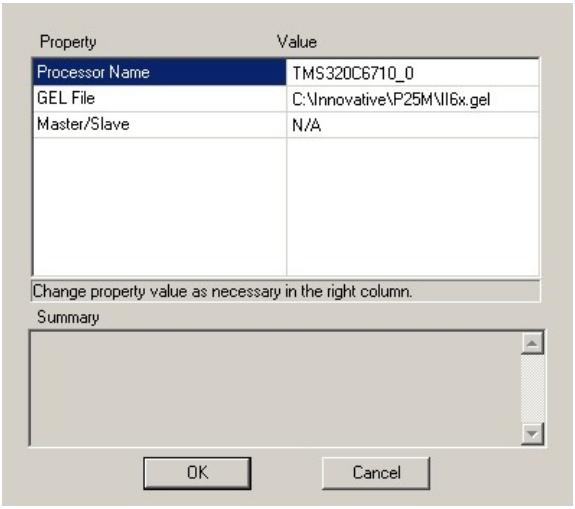


Now we add a processor

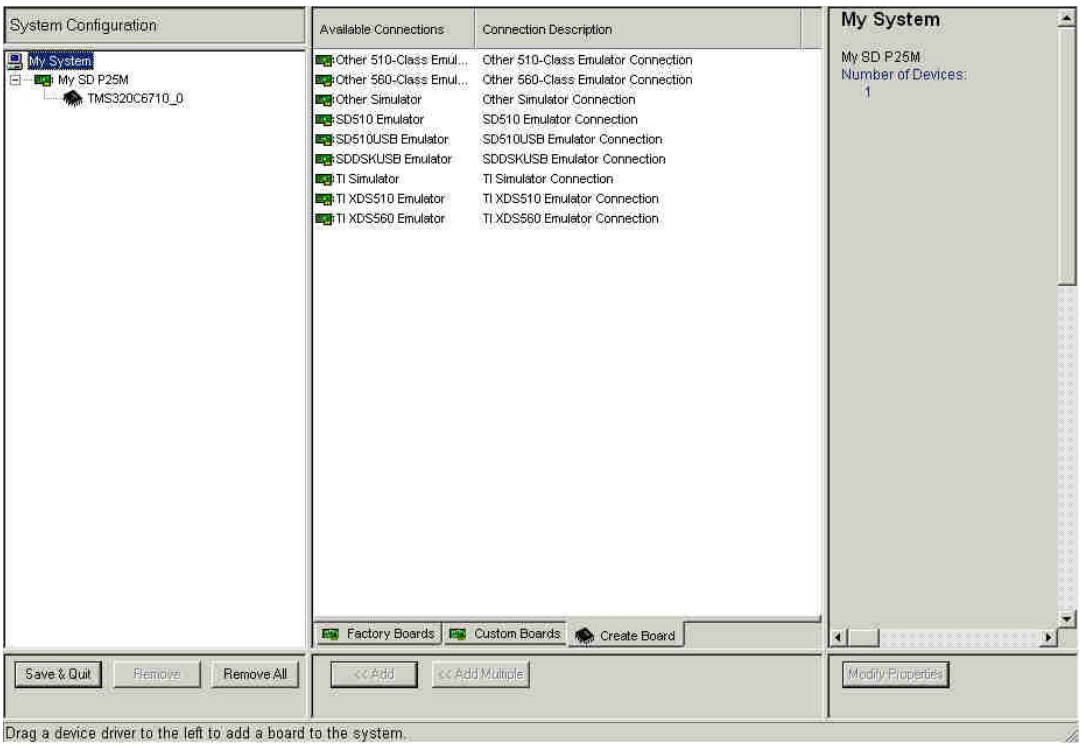
Each if the II boards have different processors so match up the closest one for your board.



Use the property sheet to find the Gel file from Innovative for your specific board.



Your system will look similar to this. Save the configuration and quit.



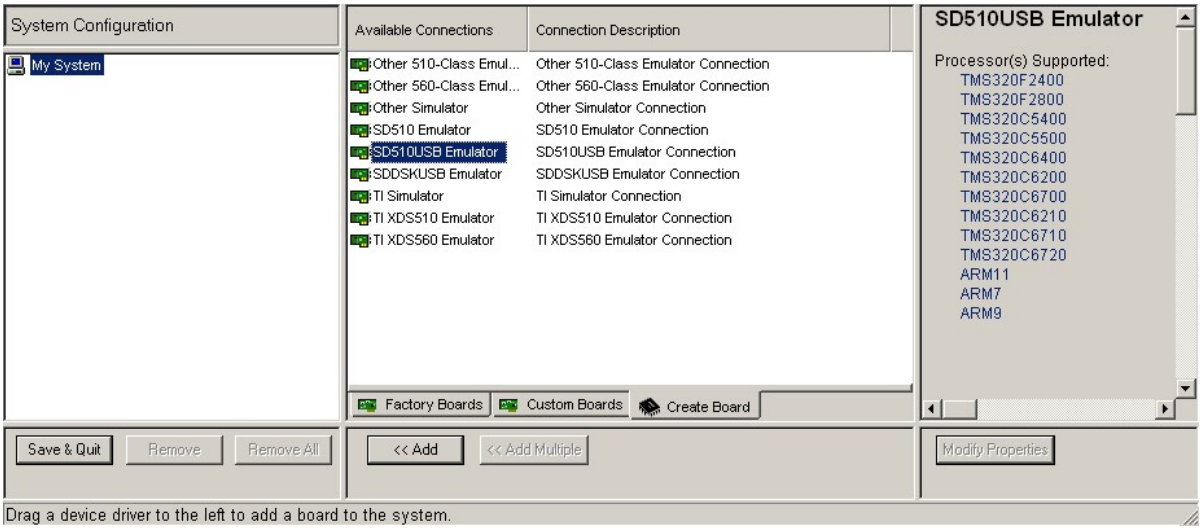
Setting up for Multi Processors with Spectrum Digital USB Jtag.

For the multi-processor setups use the following type setup. This includes the SBC6713e, Quadia, Q6x type Innovative boards.

The SBC6713e board shown will be similar in setup with the other boards. The differences will be in the types of processors and the number added.

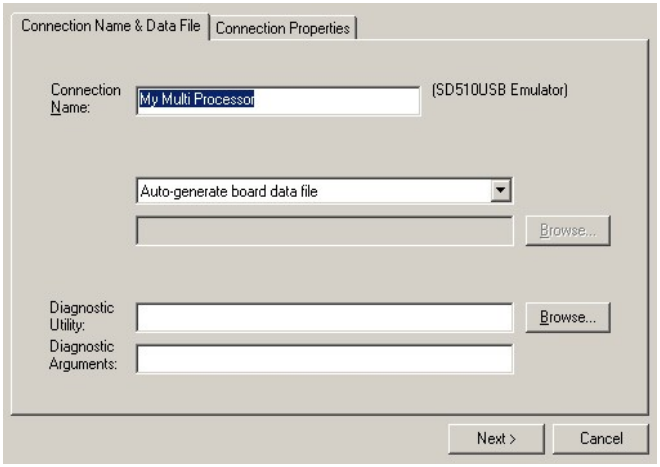
First remove any previous setups in the CCS Setup application.

Add one of the USB SD type driver.



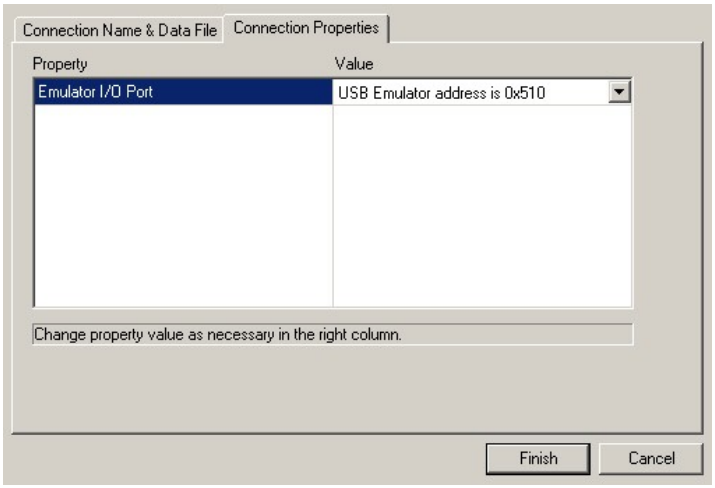
You will see the following screen.

Fill out the name of the board you are using, this can be any name you like.



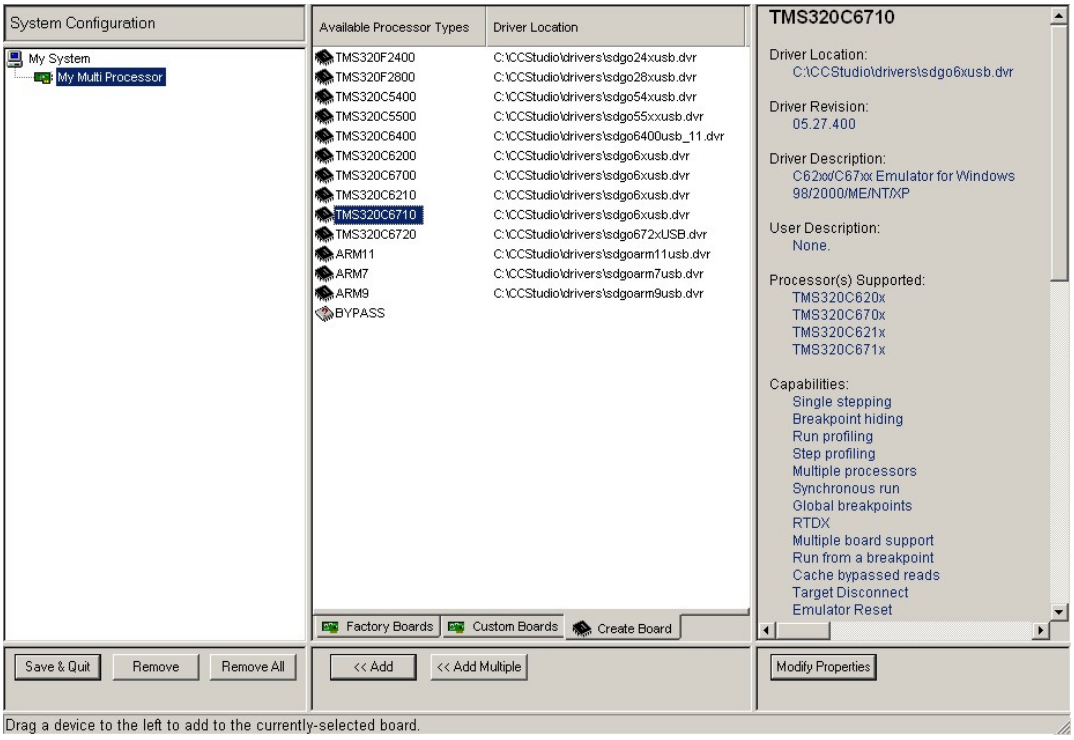
Hit next or move to the next tab

This address should match up with the address in the SdConfig.exe utility

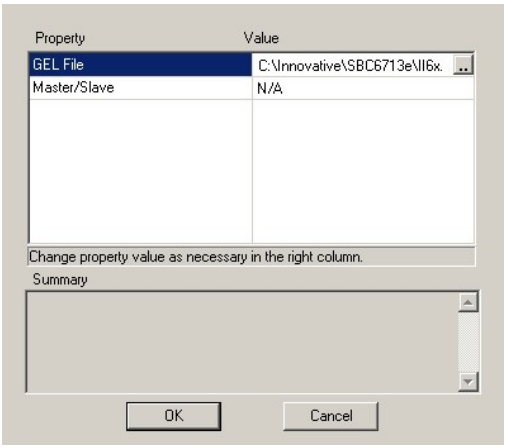


Now we add a processor

Each if the II boards have different processors so match up the closest one for your board.

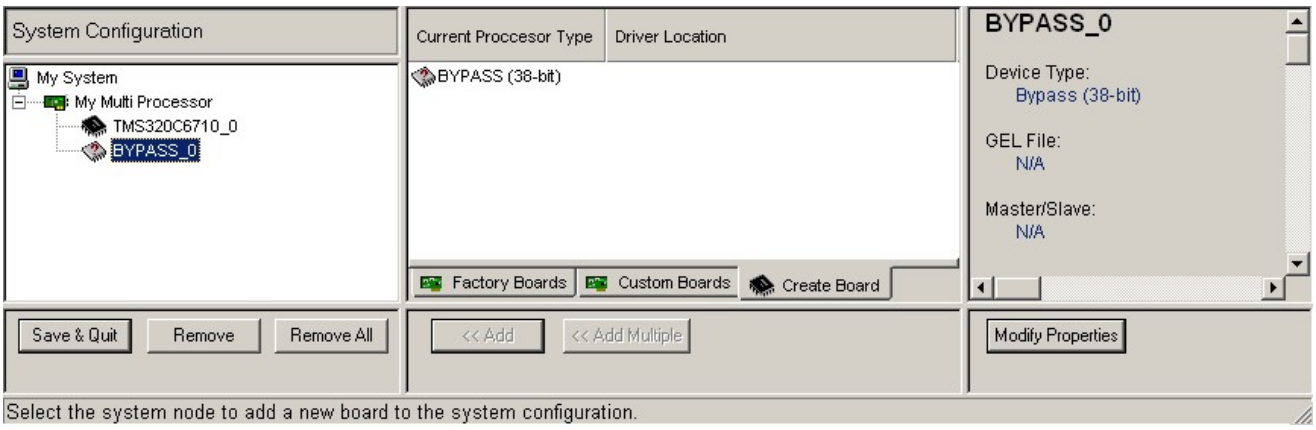


Use the property sheet to find the Gel file from Innovative for your specific board.



Close the processor and choose another processor. This will be a bypass for the DM642. Set the bypass for 38 bits. (For TMS6713 bypass use 42 bits on the first processor, the second processor will be a 64xx and the gel file from II for the DM642). For the Quadia use another C6400 type processor totaling 4 processors. All 4 will use the same GEL file from II.

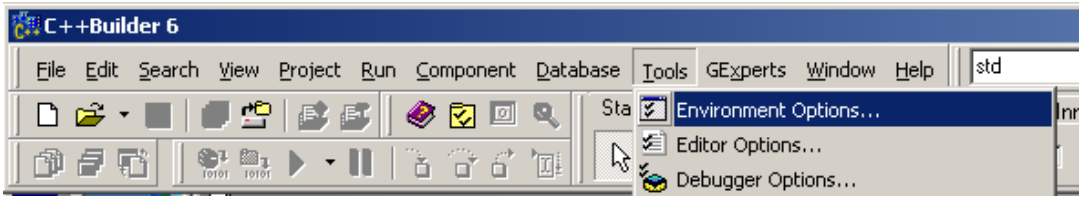
Your system will look similar to this. Save the configuration and quit.



Borland Builder Setup and Use

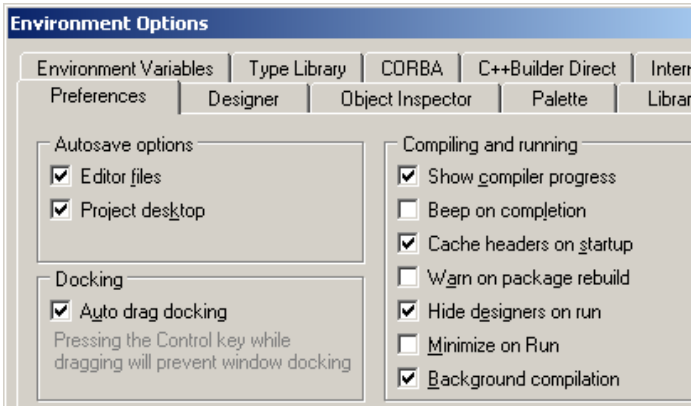
Following the normal installation of the Innovative Integration toolset components, numerous VCL components and C++ classes are automatically added to the BCB IDE. Additionally, Innovative recommends that the following IDE and project options be manually changed in order to insure simplified use and proper operation:

Automatic saving of project files and forms during debugging



31) Select Tools | Environment Options... from the main BCB toolbar.

32) This will invoke the **Environment Options** dialog:



33) Click the Preferences Tab

34) Check "**Editor files**" and "**Project desktop**" under "Autosave Options" so that project files are automatically saved each time a project is rebuilt and debugged.

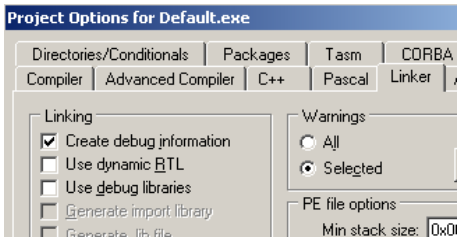
35) Click "OK"

Static-binding of built executables.

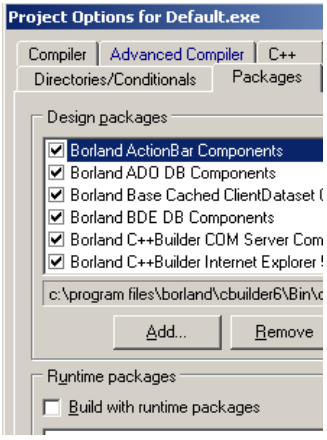
36) Click on Project | Options on the main BCB toolbar to invoke the **Project Options dialog**.

37) Click the **Linker** tab.

38) Uncheck the "Use Dynamic RTL" checkbox.



39) Next, click on the Packages tab and uncheck the "Build with runtime packages" checkbox.



These options insure that projects are built with minimal dependencies on external DLLs. See the FAQ "**What DLLs do I have to deploy with my newly created executable**" in the Troubleshooting chapter for details on which DLLs must be deployed with user-written executables.

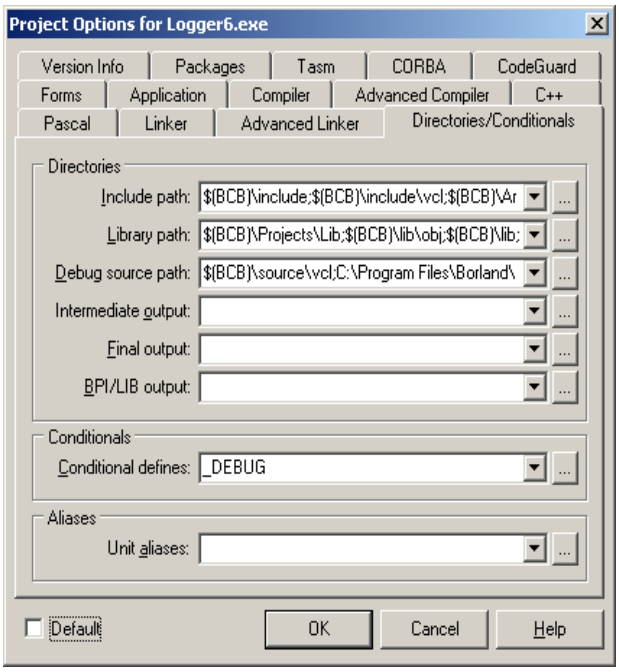
Appropriate library and include paths.

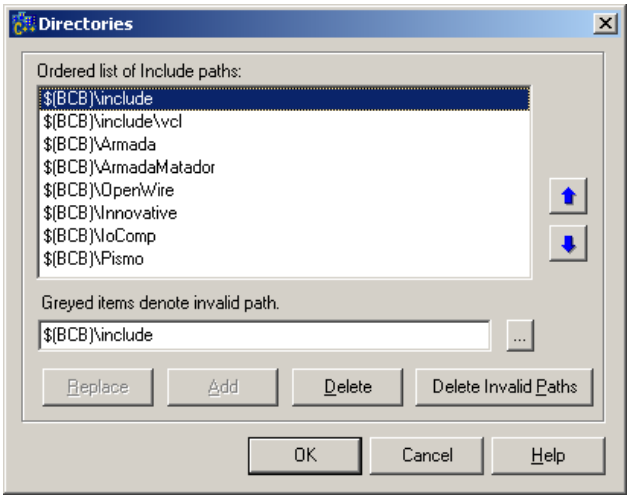
40) Click on the "Directories/Conditionals" tab.

41) Click the ellipses next to the Include Path edit box to invoke the Include Path editor dialog. Add entries for Armada, ArmadaMatador, OpenWire, IoComp and Pismo, as shown below, then click **OK** to accept these edits.

42) Next, click on the ellipses next to the Library Path edit box to invoke the Library Path editor dialog. Add entries for Armada, ArmadaMatador, OpenWire, IoComp and Pismo, as shown below, then click **OK** to accept these edits.

These changes insure that the standard Armada headers and object files are available to projects during compilation. Note that these paths may either be added to the default BCB project, by editing these options without first opening a specific project, or to specific projects after opening them. The advantage of the former is that the settings are automatically present on all subsequently-created projects.


Project Options for Logger6.exe
Version Info
Packages
Tasm
CORBA
CodeGuard
Forms
Application
Compiler
Advanced Compiler
C++
Pascal
Linker
Advanced Linker
Directories/Conditionals
Directories
Include path:
$(BCB)\include;$(BCB)\include\vcl;$(BCB)\Ar
Library path:
$(BCB)\Projects\Lib;$(BCB)\lib\obj;$(BCB)\lib;
Debug source path:
$(BCB)\source\vcl;C:\Program Files\Borland\
Intermediate output:
Final output:
BPI/LIB output:
Conditionals
Conditional defines:
_DEBUG
Aliases
Unit aliases:
Default
OK
Cancel
Help


Directories
Ordered list of Include paths:
$(BCB)\include
$(BCB)\include\vcl
$(BCB)\Armada
$(BCB)\ArmadaMatador
$(BCB)\OpenWire
$(BCB)\Innovative
$(BCB)\IoComp
$(BCB)\Pismo
Greyed items denote invalid path.
$(BCB)\include
Replace
Add
Delete
Delete Invalid Paths
OK
Cancel
Help

Chapter 4. *DSP Baseboard Overview*

Before discussing the details of software development for the DSP baseboard, a basic understanding of the components of the Malibu system and their relationships is required. This chapter provides the "big picture" view and additional information that will make the details provided in the later chapters more clear.

The Velocia Baseboard Family

Velocia family baseboards are a synergistic blend of digital signal processor hardware and data acquisition hardware. These baseboards provide a fast, flexible signal processing and data movement hardware platform with features far ahead of the competition.

The baseboard's features include:

- One or more onboard, dedicated DSP(s) for off-loading I/O processing from the host to allow data acquisition and signal processing at maximum rates. Members of the Velocia family utilize an advanced TI TMS320C6416 DSP with up to 64 MB SDRAM, enhanced cache controller, sixty-four DMA channels, three MCBSP sync serial ports and two 32-bit counter/timers. The DSP runs the royalty-free, multitasking, real-time operating system - DSP/BIOS.
- A 32/64-bit PCI/PCI-X bus host interface with direct host memory access capability for bus-mastering data between the card and host memory. The interface also supports PCI slave mode accesses from the host for configuration and application downloading.
- Flexible, baseboard-specific peripheral options options including:
- Quadia
- Two large VP40/VP50 FPGAs providing up to 10 Mgates of user-customizable logic.
- 200 MB/sec LinkPort for use in fast data transfer among DSPs .
- Rocket I/O-based SFP links for inter-board connectivity.
- Up to 512 MB shared memory which provides high-throughput local storage to the local DSP pool.
- Two PMC sites for I/O expansion.

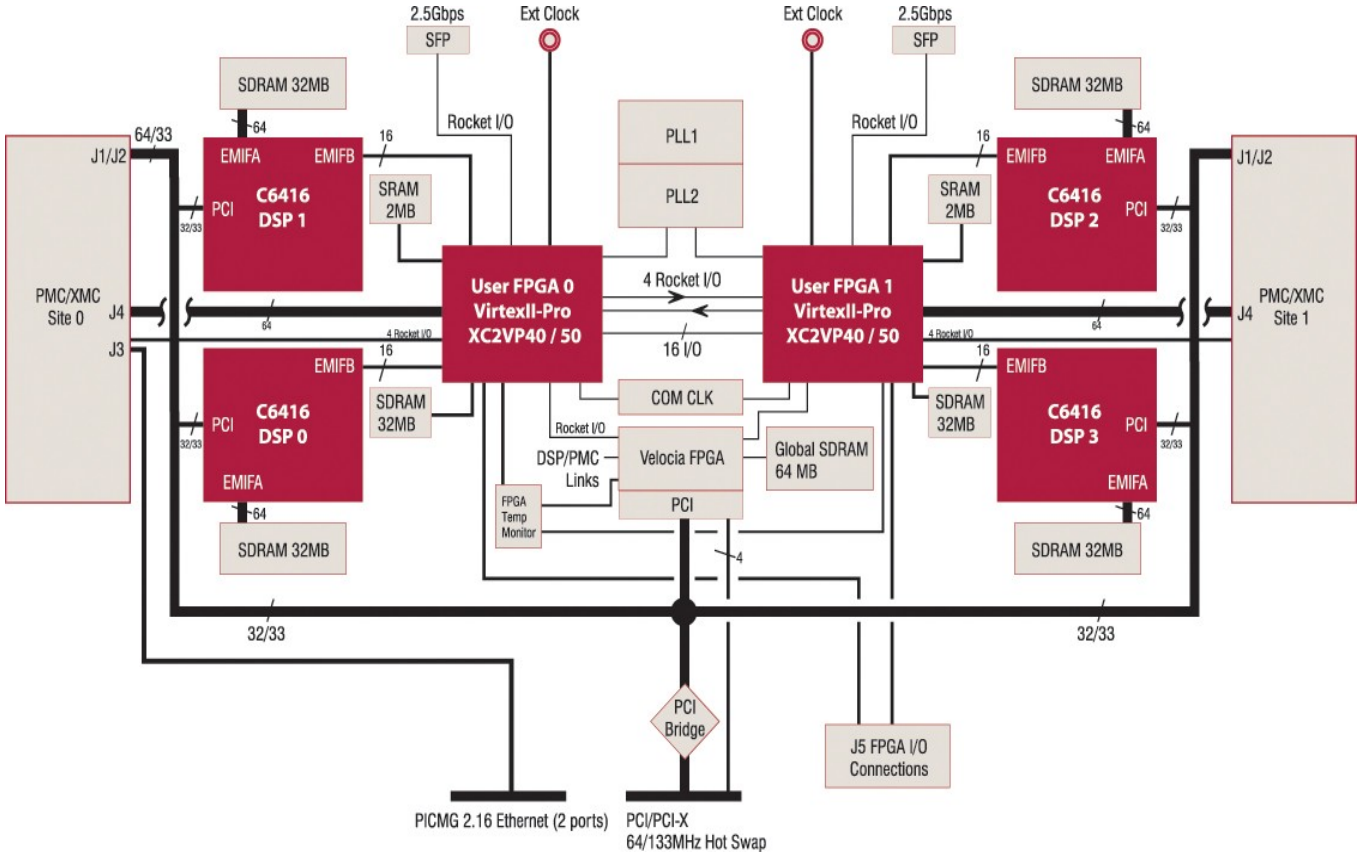


Figure 7. Quadia Block Diagram

- Duet
- Large VP40/VP50 FPGAs providing up to 5 Mgates of user-customizable logic.
- 800 MB/sec LinkPort for use in fast data transfer among DSPs .
- PMC/XMC site for I/O expansion.

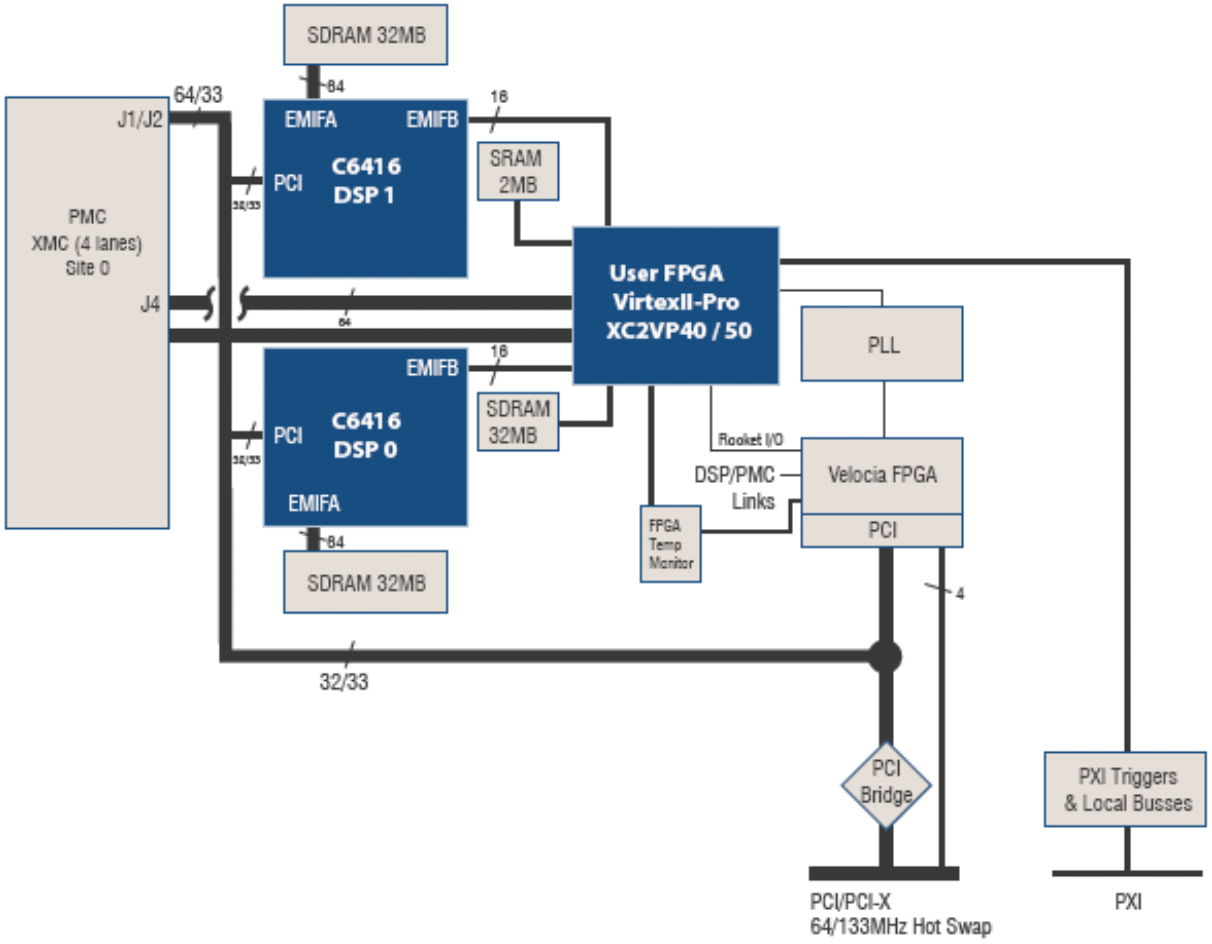


Figure 8. Duet Block Diagram

The Baseboard Device Driver

Velocia baseboards are Plug-and-Play PCI devices which require a device driver for Win2K and WinXP. The same device driver, *iixwdm.sys*, is used for all PCI baseboards. The appropriate driver is installed as part of the normal installation process for each baseboard. The driver accepts the resource assignments given for the board and configures the software to use them, making the board fully recognized by Windows.

The device driver also reserves a block of contiguous, physical memory for use as a region for bus-master transfers. This block ranges from 2-8 MB in size. A separate region is required for each DSP on each baseboard and this is permanently reserved for use by that DSP; it will not be available for Windows applications or other boards.

Under Win2k and WinXP, reserving this space may require the raising of the reserved system memory ceiling whenever:

1. One or more baseboards are installed into the PC system (including initial installation!)

2. Operating at high data acquisition rates or with large channel-count systems

An applet, ReserveMemDsp.exe, is provided to support manual adjustment of these registry properties. It must be run prior to attempting to use the DSP, run any of the supplied examples, etc. See the Applets chapter for details.

Multiple Baseboards

When installing more than one baseboard in a system, a means of uniquely addressing each installed baseboard must be employed. Windows automatically loads the baseboard device driver for each board as it initializes after PC boot-up. During this cold-start initialization, a unique instance of strategic portions of the device driver is allocated for each board installed in the PC. Correspondingly, each base board is assigned an integer identifier, referred to as the *target number,* which corresponds to the baseboard's place in the driver allocation sequence. These target numbers are assigned based on the placement of the boards on the PCI bus, so the assignment for a particular arrangement of boards is fixed unless boards are removed or re-arranged. If additional boards are added, target assignments for all boards may change.

Unfortunately, the relationship between PCI slots and driver-assigned target numbers is system-dependent. So, a means of associating a target number with a baseboard installed in a particular PCI slot is needed. To determine the slot-to-target-number associations, each Innovative baseboard has an LED that can be illuminated by a software access from the host. The included troubleshooting applet, Finder.exe, can be used to activate the LED for a specific target to determine the proper target assignments. See the Applets chapter for details.

On Quadia, each of the four DSPs (two for Duet) enumerates individually and is controlled via a device-driver which is independent of the baseboard. Due to the predictable nature of the enumeration process, the four DSPs (two for Duet) will always be assigned target numbers in direct relation to the baseboard target number. Consequently, applications can reliably discern which DSPs are associated with any given Quadia or Duet baseboard.

Slave Accesses

All baseboard and C64x DSP peripherals on the Quadia or Duet can be accessed from the Host CPU using PCI slave-mode accesses. In this mode, the Host CPU can perform 32-bit fetches or stores to any decoded memory region in the target address space to obtain or change their current value using the device's dedicated PCI bus interface. Since all target peripherals are memory-mapped, this allows the Host CPU to read or write any target peripheral register or memory location. This mechanism always works for writes, and is usable for reads *and* writes any time after the baseboard DSP is has been booted and is running a target application.

Despite their flexibility, slave accesses are unsuitable for applications requiring high-speed data transfer between the host and target. Firstly, slave accesses are non-deterministic, since they are called by Windows applications which are routinely pre-empted. Also, they are intrinsically rate-limited, since they are implemented as individual Host CPU read/write operations rather than efficient, hardware-driven bulk transfers. Nevertheless, slave accesses are invaluable in downloading target application code and performing other low-bandwidth, asynchronous I/O.

In the Malibu toolset, the baseboard provides a public *HpiEngine* object featuring *WriteBlock* and *ReadBlock* methods for slave mode accesses, so that detailed knowledge of peripheral register addresses and bit patterns is unnecessary.

Data Transfers

To address high-bandwidth data transfer applications, the baseboard is capable of high-speed transmission and reception of data via the PCI bus, using a mechanism called *bus-mastering*. When bus-mastering, the target DSP, which must be running a downloaded DSP application, transfers data between target DSP memory and Host PC memory automatically with no host CPU intervention. Since the Host CPU is not directly involved in data movement in bus-mastering mode, it is much more efficient and deterministic than slave accesses.

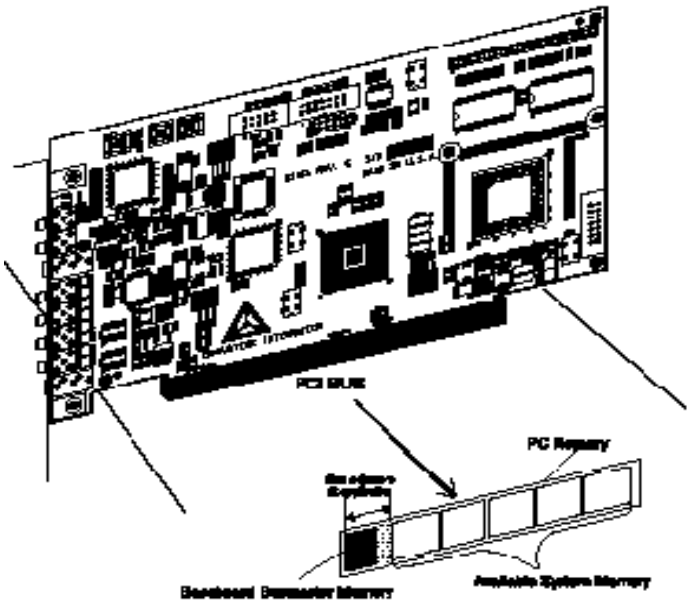


Figure 9. Bus-mastering efficiently transfers data between target and host memory

The bus-mastering interface supports sporadic or continuous acquisition and/or playback from multiple channels, simultaneously. This facility is used when performing high-bandwidth operations, such as acquiring millions of samples per second from an A/D input channel on the target DSP board.

Bus-mastering input is logically independent of bus-mastering output. It is possible to acquire data from any number and mix of input devices at a programmed rate. Simultaneously, data may be streamed out to a variety of output devices at a different programmed rate. Data flow is fully controlled by use of device drivers called from within the DSP target application.

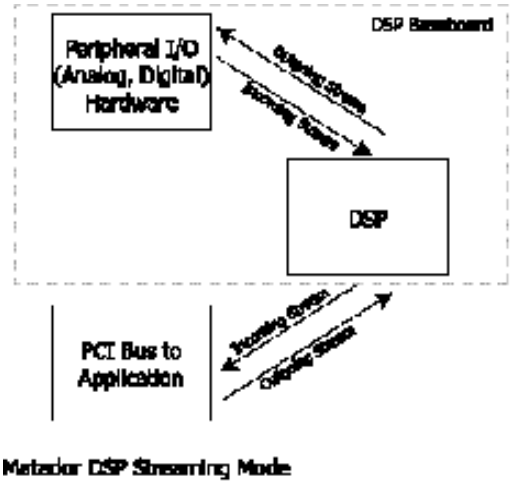


Figure 10. Bus mastering transfers are always initiated by the target DSP or PMC

During data bus-mastering, data flows between areas of page-locked host memory (specifically allocated to each and every target) and the memory of a dedicated, on-board, digital signal processor (DSP). The dedicated DSP can optionally process data as it travels between peripherals and the host application, unburdening the Host CPU of signal processing tasks providing enormous flexibility and yielding extremely high performance.

Message Packet I/O

In many applications, there is a need for additional, low bandwidth channels in addition to a high-rate data stream. Velocia baseboard DSPs support the asynchronous interchange of low-bandwidth data in conjunction with high-bandwidth bus-mastering mode I/O. Messages packets consist of a command code and channel number plus up to 14 additional 32-bit parametric data values. Messages may be asynchronously transmitted and received from any number of distinct channels by any number of threads running on both the target DSP and Host PC. Message transfers have no deleterious effect on data bus-mastering and consume virtually none of the bandwidth of the DSP, so they may be freely used even in conjunction with full rate data bus-mastering.

Class Libraries

Malibu

Malibu is the name given to the collective software suite for controlling baseboards, manipulating data streams and for aiding in the analysis of data. The details of this suite will be more fully discussed in later chapters. In this section, we will give a general view of how the software relates to Velocia products.

The Malibu suite shields the user from the nitty-gritty details of responding to asynchronous notifications of stream data and message reception, stream data requirements and message acknowledgments. Instead, a set of special C++ software class

objects, have been created to model each portion of the system. By employing software objects which model the true physical layout of the system, we can make a full-featured system more understandable.

To illustrate this, imagine that you are using a C64x DSP board within an application, such as that available on the Quadia or Quixote baseboards. Malibu contains a software component for the board, *Innovative::C64xDsp*. This component compartmentalizes all properties and functions necessary to reset, boot and download code to the DSP. Plus, the component provides properties to read and write via PCI (for slave I/O) and events (callbacks) to respond to conditions, such as when the baseboard delivers data to the PC for analysis or display. All of these features are controllable via properties and methods of this baseboard object. The functionality of other baseboards with different capabilities which are supported by Malibu is irrelevant since their functionality is isolated into other software objects. There is no need to sift through options that do not apply to your configuration. Malibu's object-orientated nature provides this benefit automatically.

Chapter 5. *A Tour of Malibu*

The purpose of this section is to provide an introduction to Malibu and a walk-through of the development process for applications developed using the Malibu library. It is assumed that MSVC or Borland BCB and Malibu are already installed and operational. This chapter does not attempt to cover general C++ programming or the use of MSVC or BCB.

Class Groups In Malibu

The classes in the Malibu suite fall into several functional categories. These are implemented within different library files within Malibu to provide maximum autonomy and keep the grouping clear. These categories are outlined in the table and the following paragraphs.

Category	Purpose
Analysis	Provide access to the common signal processing functions such as filters and FFTs; Logging and playback of waveforms and other classes needed in data acquisition and control applications. Implemented within the MalibuLib project.
Utility	Wide variety of common helper classes to manipulate elementary objects such strings, buffers, threads, semaphores and mutexes; perform file I/O; accurate timing measurements and delays; implement inter-thread callbacks. Includes a C++ implementation of OpenWire pins for interconnecting objects to form pump chains to allow automatic data processing of a data stream. However, all objects also provide functions that may be used from within the application as a stand-alone operations. Implemented within the MalibuUtilLib project.
Hardware	Provide software interface to DSP baseboards. Provision for COFF file downloads, message I/O and bus-mastering data transfers between target DSP and Host PC. Provide software interface to PMC modules. Provision for peripheral initialization and bus-mastering data transfers between target DSP and Host PC. Implemented within the MalibuMatadorLib, MalibuNetLib and MalibuMatadorLib projects.

Malibu uses C++ namespaces to distinguish its classes and methods from those of other libraries. The majority of the classes within Malibu reside within the Innovative namespace. Another common namespace is the OpenWire namespace for classes that make up the Open Wire data transfer and connection library in Malibu. There are other namespaces in Malibu that are used internally and not usually involved at the application level.

Like any C++ library, to use Malibu objects you must include the appropriate header that defines the structure of the object and its methods. If this object is in a namespace, the class name has to include the namespace to provide the full name of the class. For instance:

```
#include <Quadia.h>
...
MyClass::DoWork()
{
    Innovative::Quadia Dsp;
    Dsp.Target(0);
    Dsp.Open().
    ...
}
```

Since Quadia is in the Innovative namespace, its fully qualified name is Innovative::Quadia. To avoid having to include the namespace, a using directive can be used to tell the compiler to search the Innovative namespace automatically:

```
#include <Quadia.h>
using namespace Innovative;
...
MyClass::DoWork()
{
    Quadia Dsp;
    Dsp.Target(0);
    Dsp.Open().
    ...
}
```

These directives should be used with caution, since names shared in two namespaces may create errors in compilation.

Refer to the Malibu.chm on-line help file for detailed descriptions of any of the classes or components in the Malibu library suite.

Utility Classes

In order to provide the main services of the Malibu library, a number of building-block classes and methods were developed. Many of these classes have uses in the user application as well as in the library.

Events and Event Handlers

It is often the case in a complicated library that a procedure in a library may have to be customized for a particular application or that the application will need to be notified of certain events in a procedure.

An example of the former case is data processing. The Malibu library contains means for getting messages and data from a target baseboard, but it obviously has no way of knowing how the application wishes to process the command. In this case, the application needs to insert custom code in this place to complete the process.

An example of the latter is progress messages. If a process such as COFF downloading or Logic downloading takes a considerable time, an application may wish to display some feedback to the user giving the current progress. An event can perform this notification as part of the download process.

In order to support Event callbacks, a class needs to create an instance of the OpenWire::EventHandler template. The template parameter is the Event data class, which is the parametric information passed into the installed callback handler when an event is called. The application provides a handler for an event by calling the SetEvent() method.

Windows Synchronization
One additional aspect of Event callbacks involves the Windows UI functions. An event handler often is triggered in a different thread than the main Windows thread. This allows long tasks to work without interfering with the responsiveness of the main program, but leads to a problem if any of these handlers wish to update a Windows UI element from a handler. If the call was made from a background thread the update is not safe and can cause mysterious failures in an application.

To avoid this, an event handler can be synchronized with the main thread by using the Synchronize() method at initialization. The handler will cause the execution to be made in the context of the main UI thread, at a slight efficiency penalty.

Thread Classes
It is often useful to run tasks in a separate background thread of execution. Malibu provides a class Innovative::Thread that simplifies the creating and using of threads, as well as several derived classes that are used in Malibu for some commonly used variants. For example, StartStopThread adds the ability to freeze a thread by command and the ability to wait on several conditions.

Multi-threading Utility Classes
When using threads, you often need to have thread safe ways to signal a thread, to provide mutual exclusion a resource or code, and to wait for a condition to be signaled. Windows has these facilities in Events, Mutexes, Semaphores, Critical Sections and the WaitForMultipleObjects() API function. Malibu includes classes to simplify the use of these Windows features.

Buffer and Message Classes
Much of the processing in applications using Malibu involves data sent to and from target baseboards. Classes representing buffers and message packets have been defined to streamline the management of data.

MatadorMessage encapsulates the small 16 word message format used for command I/O on Matador baseboards and C64x DSPs.

There is an entire family of classes to manage data buffers. Each type uses a different data format. For example IntegerBuffer manages a buffer of 32 bit integers. Each buffer class has an associated DataAccess class to provide multi-threaded locking

Buffers with headers, such as those used by Packet Streams, use the PmcBuffer and PmcDataAccess classes.

Hardware and Hardware Support Classes

A major part of the purpose of the Malibu library is to provide easy interaction with Innovative hardware products. These products all require means of loading logic, software to CPUs present, configuration and control, and providing the transfer of data and commands to and from the board.

In the Malibu library, most of the details of these procedures is contained inside the library so that the application writer does not need to concern themselves with low level details. This means that it is possible for boards with different means of performing a function can be used in similar or identical ways by an application, simplifying the learning curve for the user.

Baseboards and PMC Modules
The DSP baseboard components listed below encapsulate the capabilities of the baseboard hardware.

Object	Product
Matador	Toro, Delfin, Conejo, Lobo, Oruga DSP baseboards
C64xDsp	TMS320C6416 DSP hosted on Quadia and Quixote baseboards
M6713	M6713 PCI DSP baseboard
Quadia	Quadia baseboard features (not including the four CPUs).
Sbc6713e	Supports the SBC6713e ethernet single board processor.

The PMC components perform the same function for Innovative's PMC Module family.

Object	Product
Uwb	Ultra Wideband PMC module
RacalPmc	“SIO” High-speed serial I/O PMC module

Baseboard objects are created in a one to one relationship with hardware. To associate a baseboard with a hardware device, each device in a system is given a unique index, known as the target number. These indexes are unique for each type of baseboard. Once the target number has been assigned, the baseboard can be attached to the hardware with an Open() command. If the target is not present, this method will throw an exception. Otherwise, the baseboard is ready for use. To detach from hardware, use the Close() method.

Baseboard objects also have methods to allow access to the features of the board. Some of these are unique to a particular baseboard, and are implemented as simple methods. Other board features are more complex or are shared on several baseboards. These are called subsystems. Logic loading and COFF file loading are examples of subsystems.

Subsystems are implemented as an interface class that can be shared from baseboard to baseboard, even if the implementation differs internally. Each baseboard can provide the subsystems that it requires. For example, the Quadia baseboard class has interfaces to load each of the twin user-programmable Virtex II FPGAs.

Subsystem Interfaces

Interface Object	Subsystem
IUsesOnboardCpu	CPU related functions such as Booting and COFF Downloading.

Interface Object	Subsystem
IUsesVirtexFpgaLoader	User Logic Loading.
IUsesVirtexJtagLoader	Logic EEPROM Loading.

The Interface Object classes include the methods to perform the subsystem tasks, and they also include the events that can be hooked by the application in the subsystem. For example, in the COFF loading there are events that allow the intercepting of error and status messages produced during the load, and a Progress event that can be used to provide user feedback during the process.

Target I/O Streaming Classes

Data I/O between the target and the host is a major component of many applications. It is also one of the most complicated tasks, involving interrupts on both target and host, busmastering, DMA, data buffering and buffer management, among other issues. In Malibu a particular style of I/O is packaged into a separate Stream class, which when associated with a Baseboard class, can provide the methods and events needed for efficient I/O to and from the target.

Before being used a stream must be attached to a baseboard with the ConnectTo() method. Only if this method of streaming is supported on a baseboard will the ConnectTo() compile. The DisconnectFrom() method removes the connection.

A limitation on all busmaster communications that streams commonly used is that single packet size is limited to what can fit into the allocated busmaster region. This region is allocated by the device driver at startup. The maximum size this buffer can be sized to can depend on the system BIOS or Windows. In any event, it is often relatively easy to send large amounts of data in multiple packets rather than depend on a single transfer.

Stream	Usage
PacketStream	Packet based streaming, with data from separate data sources in individual packets.
TiBusmasterStream	Packet based streaming from TI CPUs with PCI bus-mastering.
BlockSteam	Matador style streaming, with no header and interleaved channels.

Innovative::PacketStream provides packet based streaming to the newer PMC cards and the M6713 baseboard. Packets may be of different sizes, the size being inserted into the packet header. A baseboard may have a number of 'peripheral' addresses that can source or consume data. Data is marked by a Peripheral ID field to allow routing according to the source or destination of the data.

By contrast, *Innovative::BlockStream* on the Toro, Conejo and Delfin baseboards are designed for analog processing and produce more typical data streams containing interleaved data from all enabled analog channels. All blocks are of uniform size, and all data is of a uniform format for that run.

The stream *Innovative::TiBusmasterStream* supports both command packets and buffers directly to the TI C64x CPU. There are no headers, and data packets may be of any size.

Analysis Classes

Vector (1D) Signal Processing Components

Common signal processing operations such as FFTs, and filters are implemented as components within the Malibu package. These operations have been implemented using the Intel IPP library for performance. The IPP library uses the full features of the Pentium processors to make analysis even more efficient.

The *Fourier* class may be used to convert signals between the frequency and time domains. Properties control the number of points in the FFT frame, from 128 to 16K points. The *InverseFourier* class performs inverse transformations (from frequency to time domain). A property is available to enable windowing of time-series input data prior to transformation using common windows such as Hanning and Blackman.

The *LowPass*, *HighPass*, *BandPass*, *BandStop*, *IIir* and *Fir* classes perform filtering operations on data blocks. Properties control the number of filter taps to be used to implement the filter, the cutoff frequencies and the sampling rate. The *Process()* method performs a convolution on a data block using filter coefficients, which are automatically calculated using the specified properties. As with the FFT component, a property is available to enable windowing of time-series input data prior to transformation using common windows such as Hanning and Blackman.

The *GaussGen* class generates random noise, distributed in a Gaussian distribution about a mean value. This mean value and its standard deviation can be changed to suit the needs of the application.

The *RandomGen* class also generates a random noise source, but with a different distribution. This noise distribution is flat, a uniform distribution between an upper and lower boundary.

The *SignalGen* class generates contiguous sinusoidal, triangular or square waves in block format suitable for consumption by other processing functions, or to be sent to target hardware as block data. A single *SignalGen* object can provide blocks of data to multiple independent streaming output channels within an application, if so desired.

Data Storage and Retrieval

Common data storage and retrieval operations from a permanent media are implemented with the following components within the Malibu package.

The *DataPlayer* class may be used to read signals from a binary data file to be sent downstream. The downstream chain could be as simple as a direct connection to a hardware output pin such as a module DAC or a baseboard output pin, or a complex chain of analysis components, each processing the data in an elaborate, application-specific manner. The component automatically fetches data from the disk as needed to sustain the real-time data flow to downstream components. A special property, *Mode*, allows continuous replay of the data contained in the file when the end-of-file condition is reached.

The *DataLogger* class may be used to store signals received from upstream into a binary data file. The class automatically stores received data blocks to disk as needed to sustain the real-time data flow from upstream components. A special property, *Ceiling*, allows capping of the total amount of data logged to the data file.

The *IniFile* class allows access to a file formatted as an INI file. Sections and elements can be written to the file, or read back from the file. This kind of file can be used for configuration parameters for an application or system.

The *IIDiskIo* class encapsulates access to a disk file for reading or writing.

The *StringList* and *StringVector* classes provide a simple way to read in text files into memory for parsing. They differ by the kind of data structure used to hold the strings of the file. The list is efficient for large files, but has poor random access ability. The vector is better for small files that are being scanned through repeatedly.

Conversion Functions
The Malibu Library contains a number of functions for converting numeric data to strings for use in applications.

FloatToString() converts a floating point value into an equivalent string representation for display in text output.

HexToBin() and *BinToHex()* perform bulk conversions of text arrays into binary equivalents or the reverse.

IntToString() converts an integer to a string representation. A radix argument allows hex or decimal output.

StringToFloat() converts a string representing a floating point value into its numeric equivalent.

StringToHex() converts a string representing a hex integer into a numeric equivalent.

StringToInt() converts a string representing a decimal integer into a numeric equivalent.

Using the Malibu Component Suite

The Malibu library is a library of standard C++ classes. Its classes are created and used in the same way that classes of the standard library or any other library are. Versions of the library are built for Visual C++ and for Borland C++ Builder. The code that interacts with Malibu classes is identical on the two versions – the differences actually come when interacting with the different APIs for the visual portion of the application.

The Malibu library provides a simple means of accessing the features of the Matador Family baseboards, and streaming data between a Host application and target peripherals. By using Malibu, you can easily process and analyze data in real-time, as it is moved to and from the hardware.

The Malibu system uses a number of classes to perform data acquisition and analysis functions. Depending on the operations to be performed, you may need a streaming class, one or more baseboard classes, analysis classes and so on. The properties of the baseboard classes are used to define the system configuration. The properties of the analysis classes and especially the connections to other analysis components are crucial in defining the data analysis.

Event handler callbacks are another major part of creating an application in Malibu. Malibu objects provide 'Events' that the user can install a handler for that provide feedback or to customize processing.

Creating a Streaming Application in Visual C++

Creating the Malibu Objects
First we will declare the necessary objects. In this case we are developing an MFC application and we have selected a dialog-based application in the Visual C++ wizard, so that we can have a visual means of laying out the main window. This is a common technique in Visual C++.

The best place for the declarations is the dialog class that was auto-created by the application wizard. Here is how the code will look like if the code if we have given the name `CAppDlg` to our dialog class:

```
namespace Innovative
{
    class Uwb;
    class Quadia;
    class C64xDsp;
    class DataLogger;
}

class CAppDlg : public CDialog
{
...
...
private:
    Innovative::Uwb *               Uwb[2];
    bool                            UwbOpened[2];
    Innovative::Quadia *            Quadia;
    bool                            QuadiaOpened;
    Innovative::C64xDsp *           Dsp[4];
    bool                            DspOpened[4];
    Innovative::TiBusmasterStream * Stream[4];
    bool                            StreamConnected[4];

    Innovative::IntegerBuffer BB2;
    Innovative::DataLogger *  Log;
...
...
protected:

    void CoffLoadProgressHandler( Innovative::ProcessProgressEvent & event);
    void CoffLoadCompleteHandler( Innovative::ProcessCompletionEvent & event);
    void MailAvailableHandler( Innovative::TiBusmasterStreamDataEvent & event);
    void PacketAvailableHandler( Innovative::TiBusmasterStreamDataEvent & event);
 };
```

In this application we will be creating several baseboard objects. The Quadia baseboard has 4 C64x Dsps on it, each of which has its own baseboard. In addition there may be 2 Uwb Ultra Wideband PMC baseboards on the Quadia. The header only contains pointers to the objects. The actual objects will be created later.

Later in the declaration are several event handler functions. Each handler has the signature of the event it handles, which is a single class that holds parameters for the handler.

Now it's time to initialize the objects. The `OnInitDialog` member function is a good place for initialization, since the dialog controls are available but the window is not visible.

```
BOOL CAppDlg::OnInitDialog()
{
    ...
    ...
    //
    //  Create devices (but don't open!)
    Quadia = new Innovative::Quadia();

    ...
    ...

    Uwb[0] = new Innovative::UwbCs;
```

```
    Uwb[1] = new Innovative::UwbCs;


    //
    //  Coff File progress events
    for (int i=0; i<4; i++)
        {
        Dsp[i] = new Innovative::C64xDsp;

        Dsp[i]->Cpu().OnCoffLoadProgress.SetEvent(this, &CAppDlg::CoffLoadProgressHandler);
        Dsp[i]->Cpu().OnCoffLoadProgress.Synchronize();
        Dsp[i]->Cpu().OnCoffLoadComplete.SetEvent(this, &CAppDlg::CoffLoadCompleteHandler);
        Dsp[i]->Cpu().OnCoffLoadComplete.Synchronize();
        Dsp[i]->SdramCE = SdramCE;
        }

    for (int i=0; i<4; i++)
        {
        Stream[i] = new Innovative::TiBusmasterStream();
        Stream[i]->OnMailAvailable.SetEvent(this, &CAppDlg::MailAvailableHandler);
        Stream[i]->OnMailAvailable.Synchronize();
        Stream[i]->OnPacketAvailable.SetEvent(this, &CAppDlg::PacketAvailableHandler);
        Stream[i]->OnPacketAvailable.Synchronize();
        }


    return TRUE;  // return TRUE  unless you set the focus to a control
}
```

Initializing Object Properties and Events
The code immediately after the constructor of the *C64xDsp* and *TiBusmasterStream* objects are to attach handlers to events contained in the baseboard and its subsystems. In the case of the *C64xDsp* object, the COFF loading interface returned by the *Cpu()* member function has the OnCoffLoadProgress event.

This event will be called during the downloading of code to the Dsp in order to give a completion percentage of the download. The handler usually updates a progress bar with this data to give visual feedback. Because this handler will update the GUI, it needs to be synchronized with the GUI main thread. This is done by the call to the *Synchronize()* member function of the event handler object.

Below that code is the initialization of the streams. Each Dsp will have its own stream object to manage . These objects have events associated with data arriving from the target. The two event handlers are attached to functions and set to be synchronized here.

This code also shows setting a property of a baseboard. SdramCE is a property that sets which addressing space on the target the SDRAM is located. For the Quadia, it needs to be initialized to 0.

In order to use a baseboard, it must be associated with an actual device. Each device in the system is given a unique index known as the Target ID. After being assigned a target number, the device can be attached to the hardware with a call to *Open():*

```
    // Open Cpus 0 and 1 & connect their streams
    Dsp[0]->Target(0);
    Dsp[0]->Open();
    DspOpened[0] = true;
    Stream[0]->ConnectTo(Dsp[0]);
    StreamConnected[0] = true;

    Dsp[1]->Target(1);
```

```
        Dsp[1]->Open();
        DspOpened[1] = true;
        Stream[1]->ConnectTo(Dsp[1]);
        StreamConnected[1]= true;
        AppendToLog("C64x Pair #0, #1 Opened...");
```

In order to perform I/O with a baseboard, a stream object needs to be connected to it. This is done by the *ConnectTo()* method. If a baseboard does not support a type of streaming, the *ConnectTo()* call will not compile.

Event Handler Code

Data comes from the target via stream event handlers. 'Mail' messages are small (16 word) packets of data intended for command and control information exchange. Two words of the message is a header that is divided into standard fields. The TypeCode field is usually used for distinguishing different types of messages:

```
//--------------------------------------------------------------------------
//  CAppDlg::MailAvailableHandler() --
//--------------------------------------------------------------------------

void CAppDlg::MailAvailableHandler( Innovative::TiBusmasterStreamDataEvent & event)
{
    //
    //  Read the mail message packet
    Innovative::MatadorMessage  Msg;
    event.Sender->Recv(Msg);

    CString Txt;
    Txt.Format("Dsp Target %d Message:", event.Sender->Target());
    AppendToLog(Txt);

    switch (Msg.TypeCode())
        {
        case kChannelInitMsg:
            {
            //TargetLogin = true;
            int Ver = Msg.Data(0) - 0x100;
            CString Txt;
            Txt.Format("Target logged in OK - Ver: %d\r\n", Ver);
            AppendToLog(Txt);

            AppendToLog("Blocks Rcvd: 0");
            }
            break;

        case kDInInfo:
            {
            CString Txt;
            Txt.Format("Ev/Buf: %d\r\n", Msg.Data(0));
            AppendToLog(Txt);
            Txt.Format("Actual: %d\r\n", Msg.Data(1));
            AppendToLog(Txt);
            Txt.Format("Burst: %d\r\n",  Msg.Data(2));
            AppendToLog(Txt);
            Txt.Format("Actual: %d\r\n", Msg.Data(3));
            AppendToLog(Txt);
            }
            break;

        case kThresholdAlert:
```

```
            {
            AppendToLog("ALERT");
            CString Txt = "Threshold Alert Rcvd";
            AppendToLog(Txt);
            }
            break;

        case kOverflowAlert:
            {
            AppendToLog("ALERT");
            CString Txt = "Overflow Alert Rcvd";
            AppendToLog(Txt);
            }
            break;

        default:
            {
            AppendToLog("Invalid DSP message received");
            }
            break;
        }

    MessageBeep(MB_OK);
}
```

The event handler argument contains parameters for the event. In this case, the event data structure contains a pointer to the stream that generated the event. This pointer is used to actually extract the message via the *Recv()* method.

Handling the packet data event is similar: the buffer is extracted using the *Recv()* method and processed. In this case the data is logged using the *LogDataBlock()* function.

```
//--------------------------------------------------------------------------
//  CAppDlg::PacketAvailableHandler() --
//--------------------------------------------------------------------------

void  CAppDlg::PacketAvailableHandler( Innovative::TiBusmasterStreamDataEvent & event)
{
    static int PacketCount(0);
    //  Since we got this message we know a buffer is available. So read it now.
    //    Buffer will be sized to fit the incoming data.
    event.Sender->Recv(BB2);
    //
    //  Find which Cpu is our target
    int DspIdx;
    for (int i=0; i<4; i++)
        if (DspOpened[i])
            {
            if (event.Sender->Target() == CaptureInfo[i].Target)
                {
                DspIdx = i;
                break;
                }
            }
    //  ...DspIdx is which Dsp # to use
    //
    //  Increment
    CaptureInfo[DspIdx].CaptureBlocks++;
    //
    //  Log the data block
    LogDataBlock(DspIdx, BB2);
    //
    //  Update message showing data arrival.
```

```
    CString Text;
    Text.Format("Dsp %d, Packet %d with %d words arrived", DspIdx, ++PacketCount, BB2.IntSize());
    AppendToLog(Text);

}
```

Loading COFF Files

Operations such as downloading COFF files to a DSP are grouped in an interface class so that the methods used to perform them and the events presented are the same from board to board. This code initiates a download to all four CPUs on a Quadia. Events can be hooked to provide feedback on the progress of the download.

```
void CAppDlg::OnBnClickedDownloadCoff()
{
    CString filename;
    CoffFileNameEdit.GetWindowText(filename);
    std::string FileName(filename);

    for (int i=0; i<4; i++)
        if (DspOpened[i])
            {
            AppendToLog("---------------------------------------");
            CString Txt;
            Txt.Format("--  COFF Load  Dsp #%d", i);
            AppendToLog(Txt);
            AppendToLog("---------------------------------------");

            Dsp[i]->Cpu().DownloadCoff(FileName);
            }
}
```

Loading Logic Files

Many baseboards have downloadable logic to provide customized behavior. Loading this logic is also grouped into an interface class. In the code below, one of the Quadia's two logic chips is being loaded. The interface class also contains events that can be hooked to provide feedback in the user interface.

```
//-----------------------------------------------------------------------------
//  BaseboardLogicLoadDialog::OnBnClickedQfpga1Cfgbtn() --
//-----------------------------------------------------------------------------

void BaseboardLogicLoadDialog::OnBnClickedQfpga1Cfgbtn()
{
    if (!Owner->QuadiaOpened)
        {
        Owner->AppendToLog("No Quadia Installed");
        return;
        }

    CString ExoFilename;
        Fpga1FileName.GetWindowText(ExoFilename);

    if (! Innovative::IIFileExists(ExoFilename))
        throw Innovative::IIException("Exo file not found!");

    Owner->AppendToLog("----------------------\r\nParsing FPGA 1");
    Owner->UpdateWindow();
    Owner->Quadia->Logic(1).ConfigureFpga(std::string(ExoFilename));
    Owner->AppendToLog("----------------------\r\n");
    Owner->UpdateWindow();
}
```

Chapter 6. *About the Baseboard*

Velocia Family Overview

All Velocia baseboards feature the Texas Instruments TMS320C6416 digital signal processor, Xilinx FPGAs and extensive peripheral feature set to support demanding signal processing applications. The tight coupling of the DSP, FPGA and peripherals make these boards well suited for a variety of applications such as software digital radio (SDR), communications, ultrasound, RADAR and many data acquisition applications.

The Velocia family of DSP cards have similar, though not identical features. Since the cards are all built around the '6416 DSP and Xilinx FPGAs, many parts of the card architecture are similar as is the software development kit.

Quadia/Duet Overview

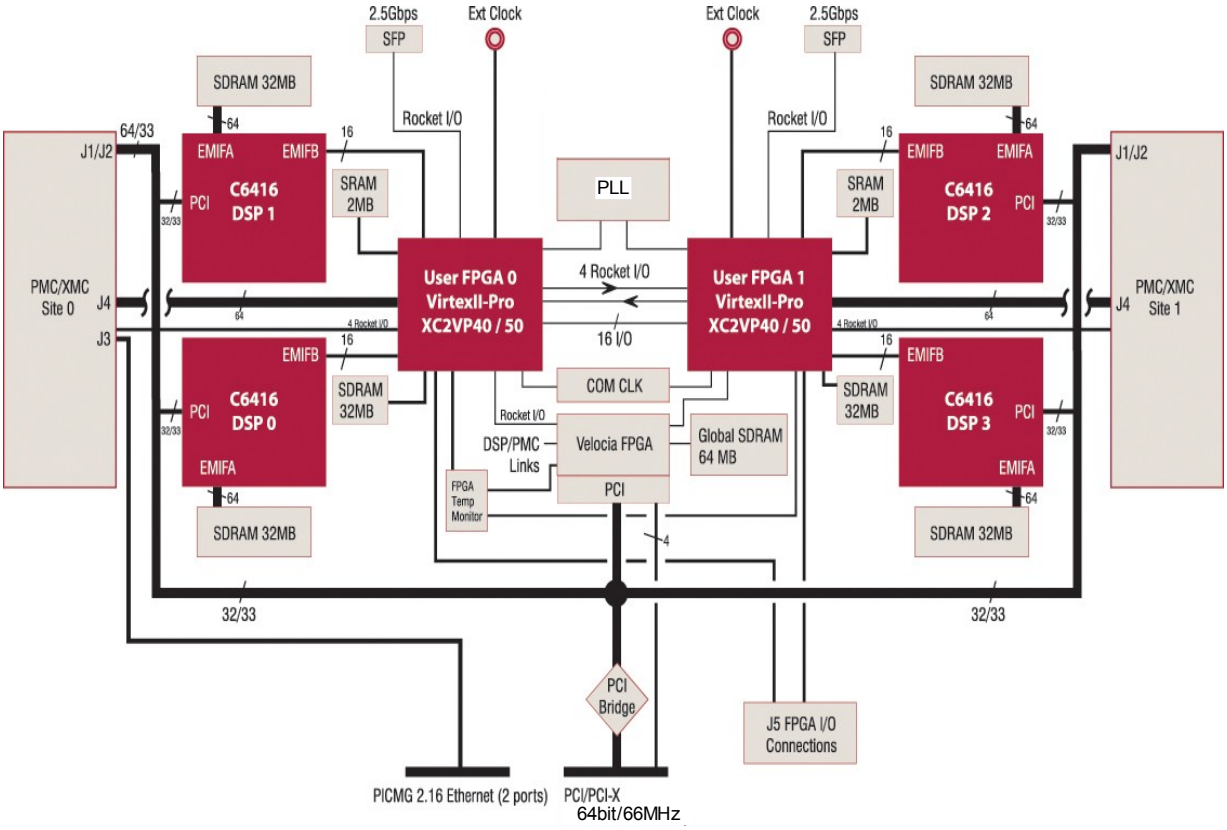
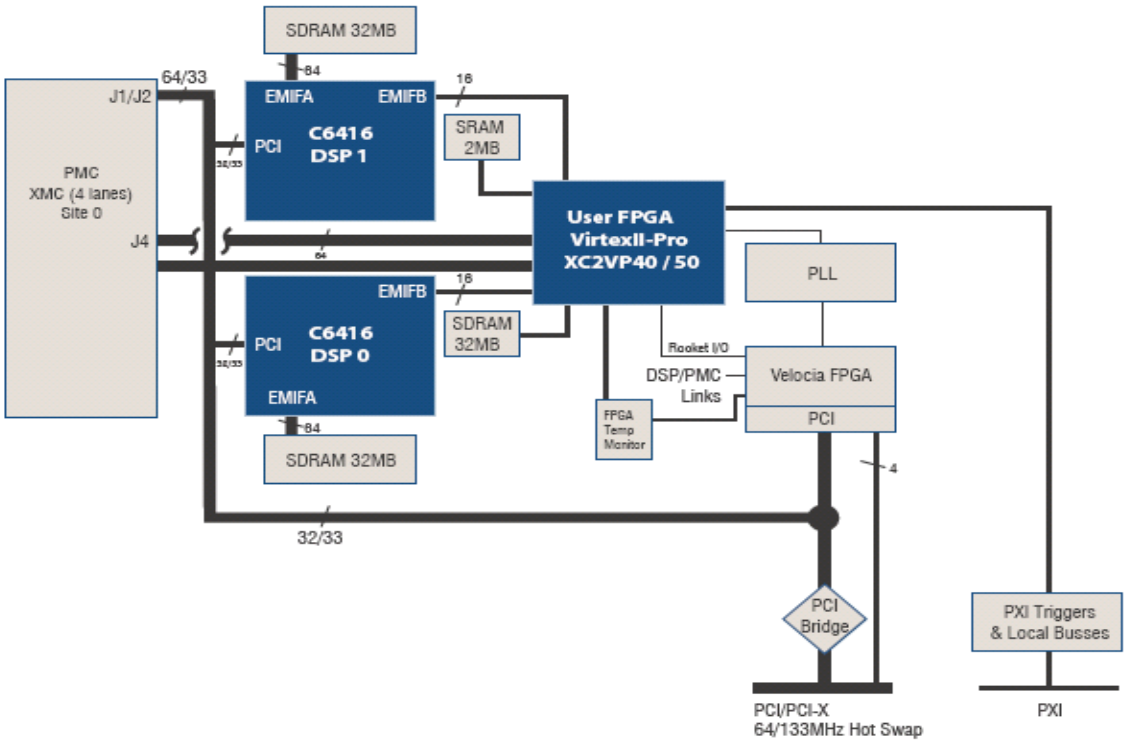


Figure 11. Quadia Block Diagram

Quadia is built around four Texas Instruments TMS320C6416 digital signal processors (Two DSPs for Duet) chips coupled with Xilinx Virtex2 Pro FPGAs (Single for Duet) as the computational engines. The DSPs and FPGAs, with their local memories, form computing cores that combine speed and flexibility for a powerful signal processing platform. Tight integration of the computing and IO, using PCI and Rocket IO, with PMC/XMC IO modules allows Quadia and Duet to support high speed, real-time signal processing.

Figure 12. Duet Block Diagram



Note: Duet is essentially "half a Quadia", with one FPGA, two DSPs and a single PMC site. The differences from the Quadia are detailed at the end of this chapter.

The four DSPs (two for Duet), operating at up to 8000 MIPs, are complemented with dual (one for Duet) Xilinx Virtex2 Pro FPGAs. The Virtex2 Pro has many features for both signal processing and high speed computing such as a embedded multipliers, dual PowerPCs, gigabit serial ports, embedded memory, and a logic fabric of 4 M gates (approximate for standard VP40).

Both Quadia and Duet features a flexible data plane and PCI architecture suited for high rate, real-time signal processing. The data plane provides high speed, low latency data transfers between the DSPs, FPGAs and IO and is extensible to other cards. This data plane is complemented by a local PCI bus that is flexible and high speed, making both Quadia and Duet easy to integrate into systems and program.

Peripherals on the card include dual PMC/XMC sites, local FPGA memories, global memory pool, timing controls for synchronization and sample rate generation, and system connectivity.

Many of the features implemented on Quadia and Duet are part of the FrameWork Logic such as the data plane communications. Users can build on top of this infrastructure to incorporate DSP functions and system features using logic development system. Developers should see the *FrameWork Logic User Guide* or *MATLAB BSP Manual* for more information.

Processing Cluster

Quadia is organized as two processing clusters, each composed of two '6416 DSPs and a Virtex2 Pro FPGA with a PMC/XMC module for IO. Each computing cluster has local FPGA memory, 2MB of SRAM and 32MB of DRAM. The two clusters are identical in their features and capabilities. Duet, on the other hand, has one such cluster.

As can be seen from the cluster shown, each DSP has its own private memory on EMIF A and is connected to the FPGA over EMIF B. The private memory is 64 MB of SDRAM running at 133 MHz. The connection to the FPGA over EMIF B is 16-bit at 133 MHz, yielding a maximum rate of 266 M bytes per second. The local PCI bus connects both of the DSPs and the PMC together, and is isolated from the system PCI bus by a PCI bridge.

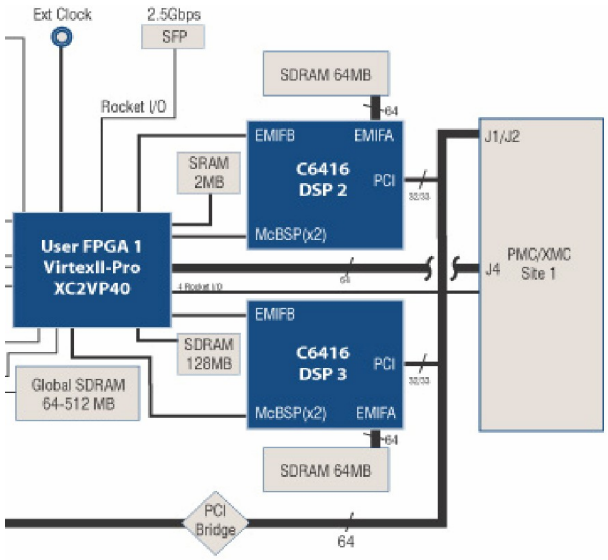


Figure 13. Quadia Processing Cluster Block Diagram

The Virtex2 Pro FPGA provides a powerful computing element in cluster with its dual PowerPC cores, dense logic array and DSP features. The FPGA in each cluster is tightly coupled to the DSPs and PMC/XMC IO module in that cluster. Private J4 and XMC (VITA 42.0) connections to the PMC/XMC allows modules to implement high performance control and data interfaces to the PMC IO. The FPGA also has private memory comprised of 64 MB of DDR SDRAM and 2 MB of synchronous ZBT SRAM for use by the FPGA or PowerPC devices embedded in the FPGA.

Connectivity outside of the cluster consists of the PCI bus, the data plane connections from the FPGA and the SFP port for off-card connections for system expansion and IO communications.

Connectivity

Both Quadia and Duet has two primary connectivity layers: the PCI bus and the data plane. These two communications layers serve complimentary purposes that allow the architecture of Quadia and Duet to have independent data and control buses required by many applications.

PCI Buses

The local PCI bus provides system-level connectivity to the on-card DSPs, FPGAs and other resources over a local PCI bus architecture. The PCI bus is used primarily for system integration such as command, control and data transfers to the host.

The following diagram shows a simplified view of the Quadia PCI architecture. (Only PCI buses are shown, not all connections, for clarity.)

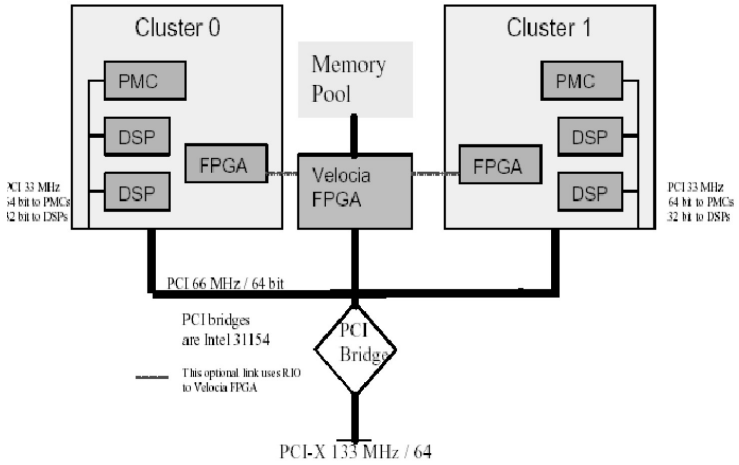


Figure 14. Quadia PCI Architecture

As can be seen from the diagram, all resources are positioned on the *local bus*, behind a bridge to the system PCI bus. The DSPs, PMC modules and Velocia FPGA all enumerate on the PCI bus of The host. This allows the host software to directly communicate with each device on the Quadia/Duet local bus, making device communications less complex and more intuitive to the programmer.

Quadia can be used on host PCI buses running at up to 66 MHz and 64 bits. Host buses running at lower rates are automatically accommodated by the bridge. The host bus may be 5V or 3V signaling. The PCI bridge is an Intel 31154, a popular device that has native support by Windows and is widely used. Devices on the local bus run at 32-bit, 33 MHz because the PCI interface of the DSP is limited to that speed, even though the PMC modules and Memory Pool may be capable of 66 MHz, 64 bit operation. The local bus uses 3.3V signaling.

PCI 64 bit/66MHz

The cluster FPGAs have an optional connection to the PCI bus using Rocket IO links to the Velocia FPGA. These may be used to connect the logic and its PowerPC cores to the PCI bus.

A serial ROM on the card for each bridge is used to describe the bridge behavior. See the Intel 31154 user guide for more details and contact for Innovative technical support for more information on these custom configurations.

Data Plane

The purpose of Quadia and Duet data plane is to provide high rate, deterministic, low-latency data paths between devices on baseboard and to other cards or IO device. Quadia and Duet architecture has been designed to provide a private data path to all IO devices to provide the connectivity required by real-time applications. The data plane is separate from the PCI because the PCI bus is in most cases used by many devices, rendering its real-time performance for latency and determinicity unusable for demanding applications where data rates are high and real-time performance is required.

The following diagram shows a simplified view of the Quadia data plane. Note that Duet is composed of only one such cluster.

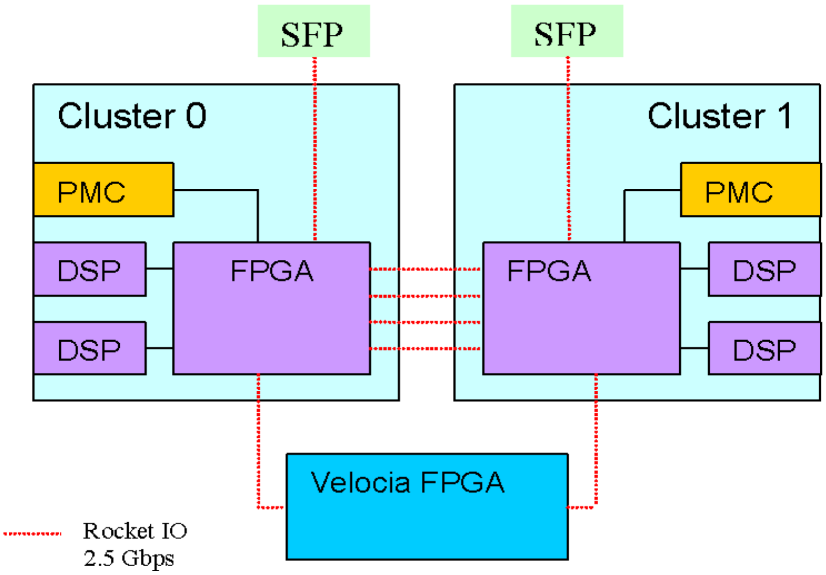


Figure 15. Quadia Data Plane Connections

The simplified diagram shows the data connections on the baseboards that comprise the data plane. The red links are Rocket IO links between the Virtex2 Pro FPGAs. The black lines from the DSPs are the EMIF B connections to the DSP. The XMC/PMCs can also communicate over the private J4 links to the cluster FPGAs. Together, these links provide fast communications for data between the DSPs, FPGAs and IO devices.

The Rocket IO links are serial connections capable of 2 Gbps (3.125 Gbps possible on special orders), full duplex, that are part of the Xilinx Virtex2 Pro FPGAs. Components are provided to customize the Rocket IO connections in the FPGAs for protocol, flow control and data rates.

The two Small Form-factor Pluggable (SFP) modules on Quadia (Duet has no SFP) provide connectivity to other cards in the system or external IO devices. System level expansion is supported by connecting directly to other processing cards, including additional Quadias or IO devices. The SFP modules support either fiber or copper physical interface to the Rocket IO ports. These are industry-standard interface modules that are available from many vendors and support long or short haul connections. The SFP modules are protocol agnostic and provide the convenience of selecting an interface module to fit performance needs without redesign.

Components in the FPGAs are also provided to integrate the DSPs and PMC modules into the data plane. It is expected that the data plane connectivity is application-dependent and that this is part of the FPGA design for that project. More details on these components are provided in the Custom Logic Development section of this manual.

Global Memory Pool

This feature is on Quadia only. Duet does not have a global memory pool.

Many applications require a large pool of on-card memory for holding data for analysis by the processors. Quadia has a 64 MB global memory pool residing on the local PCI bus accessible by any PCI device. This allows data to be shared efficiently in the pool memory by any PCI device, including the DSPs, for applications such as image processing. By placing data in the global memory, the on-card DSPs can access the data without leaving the baseboard and thus reduce overall PCI bus traffic in the system. The global memory pool provides random access into the memory and supports full PCI rates into the memory.

The global memory pool enumerates as part of the Velocia FPGA. Reads and writes to the global memory pool are 32-bit only and therefore should be 32-bit aligned. The 64 MB of memory is usable as random access memory from the local PCI. Memory write transactions are posted writes to a 1K FIFO in the logic. When a memory read transaction occurs, the Velocia FPGA latches the data address and fetches data from the memory. Data coherency is guaranteed by the memory pool control logic requiring that write transactions must be complete before a read transaction can occur. If a write transaction is in progress when a read occurs, a retry is issued to the read initiator.

No protocol is imposed by the hardware on the use of the global memory pool. Software should coordinate the use of the memory if necessary.

Timing and Synchronization Features

The baseboard has many features to generate precision clocks and synchronization signals. An on-card precision, low-jitter programmable PLL provides clocks for either communications or FPGA clocks. External clock IO from each FPGA is provided to the front panel also.

Synchronization signals for on-card or multi-card applications are provided for controlling data flows. Off-card synchronization signals are provided from each cluster FPGA. Synchronization signals between the FPGAs are also provided.

The Pismo Class Library

In order to support the baseboard as a part of a complete system, a complete set of powerful software libraries is provided to program the DSP on the baseboard and also to allow the card to interact with a host program resident on the PC. The Pismo Class Library provides support for developing applications which run on the target baseboard. The Malibu Library provides the library support for host application development.

Pismo provides extensive C++ class support for:

- Dynamic creation and runtime control of tasks
- Simplified management of and access to all TI Chip Support Library (CSL) and DSP/BIOS API functions including: Semaphores, Mutexes, Mailboxes, Timers, Edma, Qdma, Atoms, McBSP, Timebases, Counters, etc.
- Data exchange using RTDX Streaming I/O
- Foundation (base) classes for DMA-driven device driver development
- Templatized queues
- Partial standard-template library functionality via STLPort

For example, the code fragment below uses the Pismo IntBuffer class to initialize a QDMA (quick DMA) to perform a memory-to-memory move of a constant value (0) into a 4096-word buffer (at Src), then to copy the source buffer (Src) to the destination buffer (Dst):

```
// Create a source buffer of 0x1000 integers
IIBuffer Src(0x1000);
// Initialize the source buffer with zeros
Src.Set(0);
// Create a destination buffer of 0x1000 integers
IIBuffer Dst(0x1000);
Dst.Copy(Src);
```

Simple To Use
In the same way, peripheral-specific class libraries dramatically simplify access to board-specific peripheral features. For example, the code fragment below illustrates use of the PCI communications library functions to send a buffer of data to the Host PC using bus-mastering.

```
PciTransfer Xfer;

const int Size = 0x10000;
IntBuffer Buffer(Size);
for (int i = 0; i < Size; ++i)
    Buffer[i] = i;
//
//  Transmit buffer back to host
Xfer.Send(0, Buffer);
```

Not Just for C++ Experts
Note that even if you're not a C++ maven, the code is quite clear and understandable. In fact, one of the benefits of using C++ is that while it helps to mitigate and manage complexity to support creation of larger, more sophisticated applications, it is often simply used as a "better" dialect of the C language. C++ is essentially a superset of C. As such, you may freely intermix calls to legacy 'C' functions, newly-written C functions, Assembler functions and C++ functions (called methods) within C++ programs. You need not fully understand all of the enhanced capabilities and features of C++ in order to fully exploit the features of the class libraries provided in Pismo.

Unique Feature Support for each Baseboard
The Pismo Library for each baseboard provides classes and functions to access the unique features of each baseboard. For example, the Quixote version provides device drivers for AnalogIn and AnalogOut to acquire data from the analog hardware. The Pismo software isolates the application programmer from the complexities of both the hardware and DSP/BIOS.

Digital Signal Processor

The Velocia baseboard's TMS320C416 DSP operates at 1 GHz and is a 32-bit fixed-point device. The DSP interfaces to the memory and peripherals on each baseboard through its external memory interface (EMIF), which has programmable definitions for the memory interface timing.

DSP External Memory

All Velocia baseboards provide 64 Mbytes of SDRAM memory mapped to the '6416 DSP EMIFA 64-bit memory space. This is the primary DSP memory for programs and data storage. On Quixote, this memory runs at 100 MHz using an external clock for memory. Quadia memory runs at 133 MHz on EMIF A. The initialization of the memory space defines the correct parameters for the type of SDRAM used on the baseboard, including refresh timing, and should not be modified.

DSP Initialization

For proper operation of the external peripheral on the baseboard, the external memory interface control registers must be configured prior to use of the external memory interface. Applications built under the Pismo Toolset libraries will automatically initialize the registers appropriately (using code within HdwLib\IIInit.cpp). For those customers who need to initialize the registers manually, please refer to the EMIF register initialization values within the IIInit.cpp source file to obtain the required register values. Please note that the initialization is order sensitive and should be performed in the order given in the tables below.

	Register Name	Address	Value	Use
EMIF A	EMIFA_GCTL	0x01800000	0x00012064	
	EMIFA_CE0	0x01800008	0x109103C1	Asynchronous devices
	EMIFA_CE1	0x01800004	0x000000E0	PCI FIFOs (burst)
	EMIFA_CE2	0x01800010	0x000000D0	SDRAM
	EMIFA_CE3	0x01800014	0xFFFFFF23	A/D and D/A FIFOs (burst)
	EMIFA_SDRAMTIM	0x0180001C	0x000005DC	
	EMIFA_SDRAMEXT	0x01800020	0x000D8DCB	
	EMIFA_SDRAMCTL	0x01800018	0x57338000	
	EMIFA_CE0SEC	0x01800048	0x00000002	
	EMIFA_CE1SEC	0x01800044	0x00000033	
	EMIFA_CE2SEC	0x01800050	0x00000002	
	EMIFA_CE3SEC	0x01800054	0x00000002	
EMIF B	EMIFB_GCTL	0x01a80000	0x00012064	
	EMIFB_CE0	0x01a80008	0x000000B0	
	EMIFB_CE1	0x01a80004	0x4184C81C	Asynchronous devices
	EMIFB_CE2	0x01a80010	0x4184C80C	
	EMIFB_CE3	0x01a80014	0xFFFFFF23	
	EMIFB_SDRAMTIM	0x01a8001C	0x000005DC	
	EMIFB_SDRAMEXT	0x01a80020	0x000D8DCB	
	EMIFB_SDRAMCTL	0x01a80018	0x57338000	
	EMIFB_CE0SEC	0x01a80048	0x00000035	
	EMIFB_CE1SEC	0x01a80044	0x00000002	
	EMIFB_CE2SEC	0x01a80050	0x00000002	
	EMIFB_CE3SEC	0x01a80054	0x00000002	

Table 1. Quixote ‘C6416 DSP EMIF Control Register Initialization Values

	Register Name	Address	Value	Use
EMIF A	EMIFA_GCTL	0x01800000	0x00012020	
	EMIFA_CE0	0x01800008	0x000000D0	SDRAM
	EMIFA_CE1	0x01800004	0x22624912	Not used
	EMIFA_CE2	0x01800010	0x2fe27f22	Not used
	EMIFA_CE3	0x01800014	0x22624922	Not used
	EMIFA_SDRAMTIM	0x0180001C	0x000003E8	
	EMIFA_SDRAMEXT	0x01800020	0x000D8DCB	
	EMIFA_SDRAMCTL	0x01800018	0x57338000	
	EMIFA_CE0SEC	0x01800048	0x00000002	
	EMIFA_CE1SEC	0x01800044	0x00000000	
	EMIFA_CE2SEC	0x01800050	0x00000000	
	EMIFA_CE3SEC	0x01800054	0x00000000	
EMIF B	EMIFB_GCTL	0x01a80000	0x000020A0	
	EMIFB_CE0	0x01a80008	0x20F2C3B3	Burst memory (FIFOs)
	EMIFB_CE1	0x01a80004	0x20F2C3B3	Burst memory (FIFOs)
	EMIFB_CE2	0x01a80010	0x20F2C3C3	Asynchronous devices
	EMIFB_CE3	0x01a80014	0x20F2C3C3	Asynchronous devices
	EMIFB_SDRAMTIM	0x01a8001C	0x000003E8	
	EMIFB_SDRAMEXT	0x01a80020	0x000D8DCB	
	EMIFB_SDRAMCTL	0x01a80018	0x57338000	
	EMIFB_CE0SEC	0x01a80048	0x00000027	
	EMIFB_CE1SEC	0x01a80044	0x00000027	
	EMIFB_CE2SEC	0x01a80050	0x00000000	
	EMIFB_CE3SEC	0x01a80054	0x00000000	

Table 2. Quadia ‘C6416 DSP EMIF Control Register Initialization Values

The individual reset signals for each DSP target on the baseboard are controlled by the PCI bus. The normal boot sequence is to reset the DSP, load the application software, then pulse the PCI INT signal to launch the DSP from location zero. All Velocia family cards load the DSP software using the DSP PCI bus interface. Since the normal boot time of the PC is quite lengthy, the power supplies on the baseboard are stable before a valid software image can be downloaded to each DSP. Special precautions may need to be employed when interfacing external hardware to insure the target hardware remains benign during this startup interval.

Functions for loading the software and controlling reset are included in the Malibu Toolset. See the C64xDownload.exe applet description for details.

DSP JTAG Debugger Support

Standard TMS320 family JTAG debugger operation is supported by each Velocia baseboard. An external debugger connector allows use of industry standard JTAG debugger hardware from Innovative, Texas Instruments, and other third party suppliers. The DSP is the only device in the scan path for all cards except Duet and Quadia, which has the two and four DSPs in the scan path respectively.

Software for JTAG debugging and code development is TI Code Composer Studio. The JTAG port in this case is used to control the DSP program execution and memory control.

Here are simplified views of the JTAG chain. Note that there are only two DSPs (Dsp0 and Dsp1) for Duet.

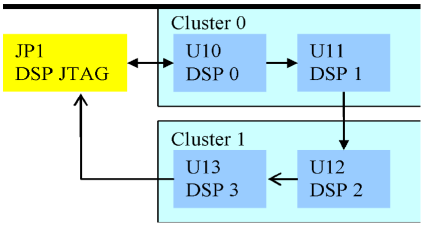


Figure 16. Quadia DSP JTAG Chain

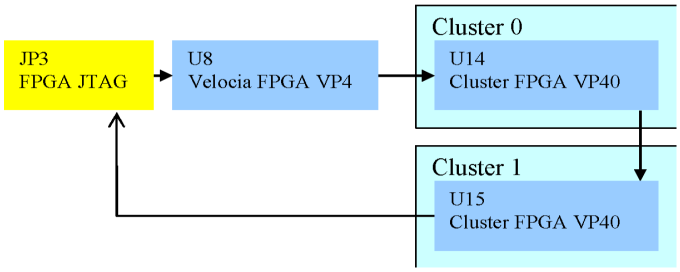
See the appendix of this manual for the connector pinout, location and type information.

FPGA JTAG Support

The Velocia cards support FPGA debug over JTAG interface. Tools such as Xilinx Impact, ChipScope and SystemGenerator use the FPGA JTAG interface as their communications and control path to the FPGA. The connector is compatible is Xilinx debug tools such as USB or Parallel Cable IV.

Figure 17. Quadia FPGA JTAG Chain (Rev F and above)

Figure 18. Quadia FPGA JTAG Chain (Rev A-E)



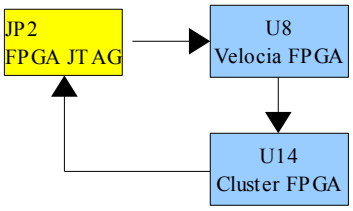


Figure 19. Duet FPGA JTAG Chain

See the appendix of this manual for the connector pinouts, location and type information.

Using the Malibu Baseboard Components

At power-up, the Velocia baseboard DSP has no program of any kind running on it. In order to have the hardware perform any action, a software program must be downloaded from the host and run. At that point, the baseboard is capable of running on its own or in conjunction with a host application using the Malibu tool set. The Malibu tool set provides special components (C++ classes) to control the initialization of the board hardware:

Baseboard	Component Name
Quadia	*Innovative::Quadia*
C64x DSP	*Innovative::C64xDsp*

Table 3. Velocia Baseboard Components

Host programs must typically instantiate one or more C64x DSP objects in conjunction with a Quadia or Duet baseboard to allow control of both CPU and baseboard resources, respectively.

Baseboards in the Velocia family contain an object of type *IUsesVirtexFpgaLoader* named *Logic* which may be used to dynamically configure the onboard Xilinx logic device(s) over the PCI bus. Use the *ConfigureFpga* method to initiate the loading of the firmware from the specified EXO file into the baseboard's Virtex logic, using the SelectMap registers mapped to the PCI bus.

IsConfigured()	Method	Returns true if the logic device has been successfully configured, false otherwise.
ConfigureFpga()	Method	Resets the logic device, then parses and downloads the specified EXO image into the baseboard logic device.

Table 4. Velocia Family Baseboard Logic Configuration Methods

The Quadia baseboard features two logic devices, whereas Duet contains single logic device. It's implementation of the *Logic* method consumes an index to support independent initialization of each device.

The *C64xDsp* object contains an object if type *IUsesOnboardCpu* named *Cpu* which implements a set of properties, methods and events that control the DSP, to allow the downloading of programs onto it, and allow the movement of data between the target and host via messaging or bus-mastering. The following table gives an overview of the initialization functions supported by this object:

Boot()	Method	Resets the DSP and all DSP-addressable peripherals, but not the Host interface. Ensures that the DSP is in a state suitable for subsequent JTAG emulation.
DownloadCoff()	Method	Resets the DSP and downloads specified COFF executable, then launches it

Table 5. Velocia Family Baseboard COFF Loading Methods

Due to restrictions in the TI C64x DSP architecture, it is not possible to successfully connect the JTAG emulator to a DSP target running a Dsp/Bios-based application program. Use *Boot* to place a processor into a benign state, suitable for subsequent JTAG emulation. Then, *Connect* the processor within Code Composer Studio. This process can be performed

automatically using the supplied C64xDownload.exe applet on any one or all C64x DSPs on a Quadia or Duet baseboard simultaneously.

To load a program onto the target, call the *DownloadCoff* method passing the name of the TI COFF executable file (.out) to download. At the conclusion of the download, the target application implicity begins execution. Be sure to start Code Composer Studio prior to downloading Dsp/Bios-based applications using this method, to avoid the problem above.

PCI Interrupt Configuration and Compatibility

Each C64x DSP residing on the PCI bus requires from at least 2 MBytes of Host PC memory space and one interrupt. Each C64xDsp instance maps two memory spaces into Host memory – one corresponding to target SDRAM memory and another corresponding to target peripherals. Additionally, a single PCI interrupt is consumed by each DSP.

Each Quadia baseboard residing on the PCI bus maps 512 MB of dual-port memory and a bank of control registers into the Host PC memory-space. No interrupt is required.

These resources are requested as part of the Plug-n-Play boot operation and are not programmable. Failure to allocate these resources will cause erratic performance. Windows will NOT report the failure to allocate resources and the user should use care during installation that the resources are available and properly allocated. The proper allocation of resources to the card may be checked on the system properties page under My Computer | Control Panel | Properties. Each instantiated C64x DSP should report that there are no conflicts.

Velocia baseboards are compatible with PCI specification revision 2.1 and have been tested with a variety of systems for compatibility. The Velocia device driver shares interrupts properly as required by the PCI specification..

In use, the Host-Side Malibu libraries handle all the details of interrupt configuration and response. The host application receives notification when data is available or required for data streaming and messaging purposes.

DSP Programming on the Baseboard

Innovative Integration's *Pismo* is a software suite allows the developer to fully exploit the advanced hardware features of the Innovative Velocia DSP product line and to reap all the benefits from DSP/BIOS. Every board peripheral has been carefully integrated into the OS and its functionality encapsulated in a device driver that can readily be controlled within DSP/BIOS applications including PCI interface, analog I/O, external bus and memory, serial ports and other I/O devices.

These drivers expose all the necessary parameters needed to efficiently control each function of the peripherals. Any peripheral board resource may be instantiated, configured and shared among program tasks. The device drivers also take care of assigning default values for unspecified or non-critical parameters of a function. C++ is used as the foundation for the Pismo libraries, but C programmers may use Pismo freely, without having to learn C++ details or C++ extensions to the C language. The C++ libraries provided in Pismo are far more capable, complete and easy-to-use than any previous generation of DSP peripheral support libraries from Innovative Integration.

Illustrative, real-time example programs are included in the software suite along with complete project files and DSP/BIOS modules. The examples act as a springboard for the development of custom, high-performance DSP application programs.

Device Drivers

Most of the peripheral devices supported by Pismo under DSP/BIOS are accessed and controlled via custom, DSP/BIOS-compliant device drivers. These drivers, provided by Innovative Integration with each DSP baseboard, are dynamically-installed as SIO - Streaming Input and Output Manager devices within DSP/Bios. Because they are constructed and installed dynamically, they are not visible within the CDB editor in Code Composer Studio projects.

Advantages of using DSP/BIOS drivers
By providing DSP/BIOS drivers for high-speed data flow between application code and peripheral devices, application programs may be developed without requiring application programmers to have detailed knowledge of the underlying peripheral hardware present on the DSP baseboard. Rather, to initialize peripherals and flow data, applications make use the stock DSP/BIOS API SIO function calls which are encapsulated within the Pismo wrapper class, Stream.

How to use a DSP/BIOS driver
Before using a DSP/BIOS driver, it must be opened for use via the `Stream::Open` method. Afterwards, the `Stream::Control` method may be used to perform any necessary device-specific initialization and/or control functions. Buffers of data are then efficiently exchanged between application and driver code via the Stream class methods `Stream::Put` and `Stream::Get` (or their lower-level sister functions Issue and Reclaim). These methods are efficient because they effect data flow via buffer pointer manipulation instead of expensive data copy operations. After use of a device is complete, it is closed using the `Stream::Close` method.

In Pismo, any peripheral devices which generates or consumes data continuously, usually on the basis of a conversion clock, is controlled using drivers for input and output classes which derive from Stream, to simplify and standardize access to features of each the streaming device driver. These objects, where available, should be used in preference to the more basic Stream base class, since the derived objects encapsulate and hide the complexity of controlling the analog, digital and PCI hardware.

The example below illustrates use of each of the above methods to open the drivers for the audio input and output codecs on the Toro DSP baseboard, set the sampling rate, echo the signals received on the analog inputs to the analog outputs, then close the driver.

```
// Desired rate of buffer cycling during streaming
const float BuffersPerSec = 10;

//--------------------------------------------------------------------------
//  IIMain() --  Loop analog input to output
//--------------------------------------------------------------------------

void IIMain()
{
        volatile bool run = true;
        volatile bool status;
        float SampleRate;

        //
        //  Terminal I/O
        cio << init;
        cio.At(Point(25, 0));
        cio << bold << "\7Echo Application\n\n" << normal << endl;

        cio << "Enter the sample rate (Hz): " << flush;
```

```
        cio >> SampleRate;
        cio << "\n  You entered: " << SampleRate << endl;

        // Instantiate the analog stream objects
        AdcStream Ain;
        DacStream Aout;

        // Simple, continuous data flow
        BasicTmb Timebase;
        Timebase.Rate(SampleRate);
        Ain.Device().Attach(Timebase);
        Aout.Device().Attach(Timebase);

        int MaxChannels = std::min(HardwareInfo()->AdcChannels(), HardwareInfo()->DacChannels());
        int EventsPerBuffer = SampleRate/BuffersPerSec;

        // Enable all analog input and output channels
        for (int i = 0; i < MaxChannels; ++i)
                Ain.Device().Channels().Enabled(i, true);
        // Size the stream buffers to signal at specified rate
        Ain.Events(EventsPerBuffer);

        for (int i = 0; i < MaxChannels; ++i)
                Aout.Device().Channels().Enabled(i, true);
        // Size the stream buffers to signal at specified rate
        Aout.Events(EventsPerBuffer);

        status = Aout.Open();
        status = Ain.Open();

        // echo input to output
        cio << "\nEchoing A/D to D/A... at " << SampleRate << " Hz\n\n" << endl;

        cio << noshowcursor << flush;
        int Count = 0;
        while(!cio.KbdHit())
                {
                cio << "\rPlaying buffer: " << ++Count << flush;

                Ain.Get();
                Aout.Put(Ain.Buffer());
                }
        cio << showcursor << flush;

        // Terminate streaming
        status = Ain.Close();
        status = Aout.Close();

        cio << "\n\nProgram terminated." << endl;
        cio.monitor();
}
```

In the example above, two device drivers are involved as data flows from the Ain device to the Aout device. Specifically, Ain and Aout are custom drivers, provided by Innovative Integration to drive the A/D and D/A devices present on the DSP board. The input device driver is named AnalogIn and the output device driver is named AnalogOut. In the example above, the analog chain configured to operate at a user-entered sample rate to flow all samples acquired from all input channels to all output channels.

Driver-specific control functions
Some Pismo drivers support special `Stream::Control` methods used to configure a device driver for a particular mode of operation, data format or other configuration or control operation outside of the scope of simple data flow. While these special `Control` methods ma be called directly, their syntax is awkward because the `Control` method does not preserve type information as it conveys data into the device driver. The wrapper classes, such as `AdcStream` and `DacStream` provide type-safe, easy-to-use methods which access to all supported, underlying `Control` functions. support control Details on available control methods for each specific driver provided in the Pismo toolset are provided in the online help files.

Driver Buffer Model
Each device driver, when opened, allocates buffers of a user-specified size (BufSize, in the example above) to be used as the destination for data samples accumulated during signal input or as the source for data samples consumed during a signal output.

DSP/BIOS device drivers implement data flow through a buffer passing mechanism. In the example above, analog sample data is continuously routed from the Ain to the Aout device via the code fragment

```
Ain.Get();
Aout.Put(Ain.Buffer());
```

which causes each of the buffers read from the `Ain` device to consumed by the `Aout` device. This is accomplished by successively passing the pointer to the data buffer most recently filled by the `Ain` driver directly to the `Aout` driver, without copying the contents of the buffer.

By default, the `Ain` and `Aout` devices are each allocated two internal buffers. Additionally, each stream object implicitly allocates one additional buffer for use in the buffer pool. Thus, each of the drivers illustrated above is allocated a total of three buffers which are managed as a rotating pool, by DSP/BIOS. If desired, the number of offers present in the pool may be modified prior to opening the driver by assigning a new value using the BufferCount method of the wrapper objects. For example,

```
// Instantiate the analog stream objects
DacStream Aout;
Aout.BufferCount(5);
AdcStream Ain;
Ain.BufferCount(5);
```

would force DSP/BIOS to allocate five internal buffers for each stream which, when combined with the single buffer implicitly allocated with each Stream object, would result in a total of six buffers in the DSP/BIOS managed pool for each device driver.

The size of these buffers may be specified explicitly, using the `Stream::BufferSize` method, or automatically calculated using the `AdcStream/DacStream::Events` method. This latter method sizes the buffers used by the streaming device driver such that they can contain the specified number of acquisition “events”, where an event is defined as one sample from all enabled A/D or D/A channels. This simplifies most buffer processing algorithms since all buffers are guaranteed to contain an integral number of samples from all enabled channels.

Generally, more buffers in the driver pool results in greater instantaneous load-carrying capacity. In practice, a larger number of pool buffers equates to a longer duration of time over which the application program can safely neglect the data servicing requirements of the device driver without risking data integrity errors. For example, in the example above the originally allocated three buffers per driver, each sized at 0x1000 bytes, running at 44.1 kHz equates to a load carrying capacity of

```
(0x1000 bytes/buffer) x (3 buffers) / (44100 samples/sec) /
(2 bytes/sample) = 139 mS
```

Whereas in the second example, with six buffers per driver pool

```
(0x1000 bytes/buffer) x (6 buffers) / (44100 samples/sec) /
(2 bytes/sample) = 278 mS
```

So, in the first example is the application program were to become busy and momentarily neglect service servicing the Ain and Aout devices for > 139 milliseconds, data integrity would be compromised and the analog output would not track the sine wave generated by the Ain driver. However, in the six-buffer example, which provides greater instantaneous load carrying capacity, data integrity would be preserved at the expense of additional memory utilization.

Driver Types

While all device drivers provided in the Pismo toolset are DSP/BIOS-compliant and accessible via the Stream class as illustrated above, there are two distinct categories of DSP/BIOS device drivers implemented within Pismo - continuous and burst.

Continuous drivers are implemented for peripheral devices which, during operation, may utilize a continuous conversion clock. Devices which fall into this category are A/Ds, D/As and codecs. Drivers written for devices of this type must be capable of sustaining continuous data flow. Special provisions may be made in specific drivers to support obtaining snapshots of the data which is flowing non-continuously, but the default mode of operation involves continuous, uninterrupted dataflow, and the driver must be capable of supporting this. For example, data which flows between an A/D converter and a buffer on the target DSP must not be suspended or inhibited for greater than one conversion sample interval, or else data loss will result and the resultant captured waveform will appear distorted on close inspection.

Burst drivers are implemented for peripheral devices which do not generate or consume data continuously during operation. Devices which fall into this category are the PCI bus interface (*PciTransfer* on the Quadia) or any other device which is capable of pacing real-time dataflow through some form of handshake mechanism. For example, data which flows between the host PC and the target DSP via the PCI bus may be sent at irregular intervals and irregular packet sizes. When either the host PC or the target DSP becomes momentarily busy and unable to exchange data via the PCI bus, dataflow is temporarily paused. However, there is no risk of data loss since the bus interface logic on the DSP baseboard provides a hardware handshake interlock which paces dataflow until both the PC and DSP baseboard are again ready for the data exchange to resume. Although dataflow is burst, aggregate data transfer rates may still be very high for burst drivers. The primary distinction is that data does not flow continuously at regular sample intervals as is the case with continuous drivers.

Pismo provides C++ base classes to enable creation of either of these driver types. Continuous drivers derive from the SioDaxDriver and SioDaxDma base classes. Burst drivers derive from the SioDriver, SioDev and SioDma base classes. Both types of drivers rely heavily on the DMA hardware present in the DSP to perform high-speed data acquisition and signal generation functions efficiently.

Driver Implementation

Both continuous and burst drivers are implemented using DMA as the data movement mechanism. The use of DMA ensures data integrity throughout the lifetime of the streaming operation. Additionally, the DMA reduces the rate of interrupt servicing by the CPU, resulting in optimal CPU bandwidth preservation.

Applications using a continuous driver are expected to service the buffers in the pool associated with that driver in real-time. That is, within the constraints of the load carrying capacity afforded by the internal buffer pool associated with the continuous driver, application code is expected to consume data from an input device or provide data to an output device at

rate nominally >= the sample rate of the conversion clock being used to drive the underlying peripheral associated with the continuous device driver.

Applications using a burst driver need not service the buffers in the pool associated with that driver in real-time. While protracted neglect in servicing the buffers in the pool associated with the driver will result in degraded throughput, data integrity is never at risk when using a burst-style driver.

DMA-enabled Drivers

While the above benchmarks are impressive, it's important to realize that the features and facilities of BIOS, used in conjunction with the silicon enhancements available in the C6000 DSPs, make DSP/BIOS-based DSP-applications less sensitive to the performance of some of these operations. For example, a combined 0.45 uS context save and restore for a hardware interrupt is certainly state-of-the-art. Hardware interrupt timings such as this are often used as a yardstick to measure the real-time performance of an embedded operating system.

But the DSP/BIOS-compliant device drivers provided by Innovative in the Pismo package fully exploit the available DMA channels in the C6000 DSPs so that hardware interrupt rates rarely exceed one KHz! The net effect is that virtually all of the bandwidth of the CPU is available for application processing. The CPU bandwidth consumed by the Innovative-supplied DSP/BIOS device drivers is minimal. Therefore, the relative importance of blazingly-fast hardware interrupt response times is decreased.

Simplified Use

Due to the relative complexity involved in programming DSP DMA channels compared to using CPU interrupt handlers for data movement, most application programmers simply avoid use of DMA entirely, resulting in highly inefficient use of CPU computational resources. In providing DSP/BIOS-compliant device drivers for all real-time peripherals within Pismo, the application programs may focus exclusively on the end-application, rather than the myriad details involved in peripheral setup, initialization and servicing. Further, use of the Pismo driver insures maximal CPU availability for application use.

For example, consider the code fragment below which illustrates all of the steps necessary to fully initialize and stream a sine wave to the audio output codec present on the Innovative Vista board at 44.1 KHz:

```
//-------------------------------------------------------------------------
//  IIMain() --  Generate a waveform on both audio channels
//-------------------------------------------------------------------------

void IIMain()
{
        const int BufSize = 0x1000;
        volatile bool status;

        // Basic I/O...
        cio << init;
        cio.At(Point(35, 0));
        cio << bold << "\7Waveform Generation Demo\n\n" << normal << endl;

        // Instantiate the analog stream objects
        DacStream Aout;
        Aout.Name("/AudioTee/AnalogOut");
        Aout.BufferSize(BufSize);
        Stream Ain("/SineAin", smInput, BufSize);

        // Open the analog I/O drivers
        status = Aout.Open();
        status = Ain.Open();
```

```
        // Start the Dds
        Aout.Control(dcSetSampleRate, 44100);

        cio << "Streaming 1 kHz sine wave to audio output driver..."<< endl;

        int Count = 0;
        // Generate waveform, send to D/As
        while(!cio.KbdHit())
                {
                if (!(++Count & 0x0f))
                        cio << "\rBuffer: " << Count << flush;

                Aout.PutFrom(Ain);
                }

        // Close the analog I/O drivers
        Ain.Close();
        Aout.Close();

        cio << "\nStreaming terminated."<< endl;
}
```

Multitasking Friendly
In addition to minimal processor loading, automatic DMA configuration and optimal bus utilization, the Pismo DSP/BIOS drivers support efficient cooperation in multitasking applications. For example, in the code fragment above, the call to PutFrom within the IIMain function will efficiently block until data is available from the Ain streaming device, allowing other tasks within the application to execute.

Analog Timebase Objects

Timebase objects provide a means to collectively configure a clock source, a start trigger and a stop trigger to control the baseboard logic which is used to pace and store the conversions of baseboard analog or digital peripherals. Timebases may thought of as external, independent physical devices like a precision oscillator timebase with programmable start/stop enables. In reality, they control one or more physical resources located on the Matador DSP baseboard. However, this portrayal of the timebase as a “virtual” clock source has advantages: For example, the Conejo baseboard contains six programmable timebases, each with different resolutions and capabilities. Which timer should be used for driving a sigma-delta converter? How are they configured when externally gating? The timebase components conceal the complexities of timebase programming by providing a separate component for each clocking technique or mode, so that you may remain blissfully-ignorant of low-level timebase initialization, routing and control mechanics.

Timebase Usage
Timebases are logical extensions to the Pismo streaming device drivers,such as AnalogIn and AnalogOut. As a stream object is created, used and finally destroyed within a target application, it performs specific driver/timebase operations at specific times. Each timebase object provides four virtual methods which are called during the lifetime of a stream object: Configure, Start, Stop and Unconfigure. These methods are perform timebase-specific initialization, and trigger functions that a stream driver automatically calls as the stream object is used. This way, application programs can be assured that these critical timebase functions are performed in the proper order and at the right time during program execution, without having to carefully code these operations within applications directly. These operations are summarized in the following table.

Stream Operation	Timebase Operation
Attach	Current timebase configuration is copied into a dynamically-created timebase object for exclusive use by driver
Open	Driver timebase is Configured - clocks routed, triggers initialized
Put/Get	Driver timebase is Started. Device-specific options are applied immediately prior to initiating data flow. For boards incorporating sigma-delta converters (Delfin), this includes application of AdcModeCtl and DacModeCtl options settings
Idle	Driver timebase is Stopped
Close	Driver timebase is Unconfigured

Table 6. Timebase Operations

For more information on timebase options and configuration, see the chapter on Analog input and output.

Interrupt Handling

In DSP/BIOS, all hardware interrupts are intended to be managed by a DSP/BIOS hardware manager. This manager allows user functions to be called as part of the interrupt process while still cooperating with DSP/BIOS. As a part of the configuration process, the user can direct the HWI manager to call a user function.

Interrupts in a C++ Environment

In a system using C++, this means of attaching interrupts leads to several difficulties. A minor problem is that of name-mangling. C++ creates a new name for every function created in order to allow overloaded functions. The DSP/BIOS configuration does not understand the new name and results in a linker error. There is a simple work-around for this:

```
extern "C"
{
    void  MyHandlerFunction( void * arg );
}
```

This declares to the compiler to create a standard C symbol name for this function `(_MyHandlerFunction)` which can be used by to the DSP/BIOS configuration tool.

A more fundamental problem is that this mechanism does not allow the interrupt handling function to be changed during the life of the program. Also, this handler function may not be a class member function. This restriction can make designing a class object that handles interrupts awkward.

The Pismo Solution

The solution implemented in the Pismo environment is to take over all interrupt handling by providing a full set of standard handlers. The user then never needs to work in the CDB editor to provide handlers. The standard Pismo handlers contain code that will call a user's installed interrupt handler function if one is provided. While this adds a small amount of latency to the interrupt, the DSP/BIOS overhead per interrupt call is still much greater and dominates the total time per interrupt.. In

general, the BIOS environment is not suited for extremely high interrupt rates. Luckily, the use of DMA to acquire data from FIFOs on peripherals means that high rate interrupt handlers are not needed.

Pismo uses a special object, a Binder, to group a handler function and its arguments in a way that can be properly called by the standard handler. One form of Binder is used to attach a stand-alone function and its arguments, another form allows the binding of an Object, a member function of that object, and its arguments. This form of binder can allow a class object instance variable to act as a handler for interrupts. Here is an example from the **Messages** example of defining a binder for a timer interrupt:

```
//
//  Timer Interrupt Handler Function
void  OnTimerFired(int arg);
//
//  Binder Object for Timer
typedef void (*IntFtnType)( int arg );

FunctionHandler<IntFtnType, int> TimerBinder(OnTimerFired, 0);
```

And attaching the binder to an interrupt:

```
//  Set up a real time clock to send commands to host on
//    Target channel...
Irq Timer0( intTimer0 );
Timer0.Install( TimerBinder );
Timer0.Enable( false );
//
//  Turn on the clock at 5 hz
DspClock  Tclk0(50.0, 150.0);

Timer0.Enable( true );
```

In the example, `TimerBinder` is an object that collects the handler function, `OnTimerFired`, and its argument, 0. This object is passed into an `Irq` object associated with the `TCLK0` interrupt. When the timer interrupt fires, the handler will be called with its argument. The binder is a template, allowing any type of argument to be used with an interrupt handler.

Class Irq

Class `Irq` is an object that can be created to manage a specific interrupt. It has functions to set, clear, enable and disable the interrupt and also allows a handler to be installed that will be called whenever the interrupt fires. In the above code, see how all functions involving the interrupt were encapsulated in the methods of the `Timer0` class object.

Interrupt Lock Classes

A common need in a program is the ability to disable a particular interrupt, or all interrupts, in a portion of the program. The standard means of standalone functions (an disable followed by a enable interrupts) has a few problems. The first is that the means does not nest well. If a function blocking interrupts is nested in a second one, interrupts will be re-enabled at the wrong time. A second is that if the function has multiple return paths, each must have the re-enable code in it. The introduction of C++ exceptions makes this problem even worse.

The Pismo library provides a set of class objects that meet this problem. These lock objects disable a particular interrupt or all interrupts in a region and restore the state to what it was on entry when the lock object is destroyed. If the object is created on the stack, any means of exiting the block in which the object is defined will cause the cleanup code to be called. Calls to these objects properly nest as well.

Lock Class	Interrupts Affected	TI Class Library
InterruptLock	One IRQ	CSL.
GlobalIntLock	All interrupts	CSL.
HwiGlobalIntLock	All interrupts	DSP/BIOS.

Table 7. Interrupt Lock Classes

Interrupt Binder Templates

The Binder system can be thought of as a more flexible and powerful version of a function pointer variable, allowing a user callback function to be called indirectly without knowing more than the interface to the function. Since the binder objects are templates, the type of the function and its arguments are not fixed but can be of any type. Also, member functions can be bound to an interrupt, which a callback function can never do.

The Binder system is powerful, yet in practice is quite simple to use. This system illustrates the power of the C++ language to contain a complicated system in a simple-to-use package.

Class InterruptHandler

This class is a base class for the ClassMemberHandler and FunctionHandler templates. It provides the interface the Pismo system uses to call the interrupt handler.

Class ClassMemberHandler Template

This template allows the binding of a member function of a class object with the object to call and an argument of any type. In this example the IsrHandler class is bound to a timer interrupt:

```
class IsrHandler
{
public:
    IsrHandler()
        : Binder(*this, &IsrHandler::MyHandler, &Tally), Tally(0)  { }
    ClassMemberHandler<IsrHandler, unsigned int *> Binder;

    void MyHandler(unsigned int * tally)
    {
        *tally += 1;

        if ((*tally & 0x7f) == 0)
            rtdx << "Isr tally: " << *tally << endl;
    }

private:
    // Data
    unsigned int    Tally;
};

// Instantiate a concrete instance of above class..
IsrHandler  Isr;
```

```
void IIMain()
{
    // Dynamically create an Irq object tripped from onchip timer 0
    Irq Timer0( intTimer0 );
    // Bind and install the interrupt vector
    Timer0.Install( Isr.Binder );
    // Program onchip timer 0 to signal at 100 Hz
    Timer0.Enable( false );
    DspClock Clock(100, 150, true, 0);
    Timer0.Enable( true );

    // Use RTDX event log to monitor progress
    rtdx.Enabled(true);

    rtdx << "Message from within IIMain,,,"<< endl;

    // Go to sleep...
    while (1)
        TSK_yield();
}
```

In the above example, the handler uses a **int *** argument to pass out information from the interrupt routine.

Class FunctionHandler Template
This template allows the binding of stand-alone function with an argument of any type. In this example the OnTimerFired function is bound to a timer interrupt:

```
//
//  Timer Interrupt Handler Function
void  OnTimerFired(int arg);
//
//  Binder Object for Timer
typedef void (*IntFtnType)( int arg );

FunctionHandler<IntFtnType, int> TimerBinder(OnTimerFired, 0);
```

This is the installation of the handler in the program:

```
    //  Set up a real time clock to send commands to host on
    //    Target channel...
    Irq Timer0( intTimer0 );
    Timer0.Install( TimerBinder );
    Timer0.Enable( false );
    //
    //  Turn on the clock at 5 hz
    DspClock  Tclk0(50.0, 150.0);

    Timer0.Enable( true );
```

EDMA and QDMA Handling

The TI C6000 processor supports a rich, powerful DMA engine to move data without CPU intervention. There are two kinds of DMA allowed. One, EDMA is full featured but can take some time to set up. QDMA is TI's facility for quick DMA movement of data. It is similar to a normal DMA transfer except that it is software triggered and performs only a single transfer. No linking of blocks is permitted with QDMA. It also is faster to initiate as only a few registers need to be set to start a new transfer.

Both kinds of DMA use a set of registers to define the configuration of a DMA transfer. By properly configuring the settings, many different transfer types can be performed, such as interelaved data, two dimensional arrays, and so on. See the TI Peripheral Library guide for more information on configuring EDMA and QDMA.

The QDMA has a single set of configuration registers, so only one QDMA may be in progress at the same time. The EDMA has a pool of blocks that may be used to define simultaneous, complex transfers.

Class DmaSettings
The DmaSettings class manages an image of the settings registers used to configure a QDMA or EDMA transfer. It provides properties to read and set the individual fields of the registers, saving the user the effort of masking bits and shifting data. It even provides functions that preconfigure some commonly used transfers, saving even more programmer effort.

The following code fragment shows how the setter functions are used to set up for a transfer. The DmaSettings class returns a reference to self on all setter functions, allowing multiple parameters to be set on a single line:

```
DmaSettings Cfg;

Cfg.Priority(DmaSettings::priHigh).ElementSize(DmaSettings::is32bit)
Cfg.SourceIncr(DmaSettings::Incr).DestinationIncr(DmaSettings::Incr);
Cfg.TCInt(true).TCCode(1).FrameSync(true);
Cfg.SourceAddr((int)&src_array[0]).DestinationAddr((int)(dest_array+50));
Cfg.ElementCount(50).ElementIndex(1);
Cfg.FrameCount(0).FrameIndex(1);
```

Class Qdma
This class manages the posting of Qdma requests. It contains functions to allow configuration of a transfer, initiating a transfer and completion notification via either an interrupt or a polling function. Because the system state is saved in the object, transfers can be predefined and saved to be posted at a later time.

As with all DMA objects, the Qdma object uses an internal DmaSettings object to define the transfer. The Settings() method provides access to the object to allow calling the DmaSettings classes own configuration functions, or configurations can be loaded from a second object with the Load() method.

```
// Q is a Qdma object, here we change the destination address
Q.Settings().DestinationAddr((int)(dest_array+0x10));
```

For QDMA, a transfer is initiated when the parameters are loaded into the QDMA registers. This is performed by the `Submit()` method, which starts the preconfigured transaction, or loads the passed in configuration and submits it.

Only one Qdma transfer may be active in the system at one time. Multi-threaded applications must arbitrate Qdmas as appropriate.

If a terminal count interrupt is not used, a call for `WaitForComplete()` will delay until the completion occurs. `TestComplete()` will return a flag that can be used to check completion without blocking.

Qdma transfers may be configured to generate Terminal Count interrupts on completion of the transfer. Which TC bit is signaled is configured in the settings block.

A user supplied handler, similar to an interrupt handler, can be associated with the terminal count interrupt by a call to the `TcIntInstall()` method. The DMA system shares a single interrupt for all TC interrupts, and the system will call the installed handler when the particular bit in the TC register becomes set. The handler installer requires an Interrupt Binder Object (See “Interrupt Binder Templates” on page 85.) as an argument to associate a handler function or method and

argument for the interrupt forwarding mechanism of Pismo. A second function, `TcIntDeinstall()` removes any installed handler.

Once installed, TC interrupts may be enabled or disabled by a call to `TcIntEnable()`.

The following example shows a full Qdma transfer with TC interrupt handling. In this example a class member function is bound to handle the interrupt response.

```
class DmaIsr
{
public:
    typedef void (*IntFtnType)(void * fallow);

    DmaIsr()
        : Binder(*this, &DmaIsr::MyHandler, NULL)  {  }

    void MyHandler(void * fallow)
    {
        qdma_not_done = false;
    }

    ClassMemberHandler<DmaIsr, void *> Binder;
};

DmaIsr Isr;

void IIMain()
{

    DmaSettings Cfg;
    Cfg.Priority(1).ElementSize(0).SourceIncr(1).DestinationIncr(1);
    Cfg.TCInt(true).TCCode(0);
    Cfg.SourceAddr((int)src_array).DestinationAddr((int)dest_array);
    Cfg.ElementCount(100).ElementIndex(1);
    Cfg.FrameCount(0).FrameIndex(1);
    Qdma Q(Cfg);

    // This QDMA operation will trip a terminal count interrupt when
    // all data has been moved.
    Q.TcIntInstall( Isr.Binder );
    InitArrays();
    Q.TcIntEnable(true);
    qdma_not_done = true;
    Q.Submit();
    while (qdma_not_done)
        ;
}
```

Class Edma
This class manages the posting of EDMA requests. It contains functions to allow configuration of a transfer, initiating a transfer and completion notification via either an interrupt or a polling function. Because the system state is saved in the object, transfers can be predefined and saved to be posted at a later time.

An additional feature of EDMA is the ability to build complicated transfers by linking EDMA transfer blocks or by chaining EDMA transfers together.

For more information on EDMA, see the TI Peripheral Guide.

As with all DMA objects, the Edma object uses one or more internal DmaSettings object to define the transfer. One block is allocated for the primary transfer, and one for each linked block. The `Settings()` method provides access to the primary transfer block's settings object. The `LinkSettings()` similarly allows to one of the link blocks's DmaSettings object. Each of these can be used to call DmaSetting's own configuration functions, or configurations can be loaded from a second object with the `Load()` method.

```
// Ed is a Edma object, here we change the destination address
Ed.Settings().DestinationAddr((int)(dest_array+0x10));
```

The EDMA transfer can be attached to one of a number of channels. To attach an EDMA to a hardware interrupt, use the channel with the same number as the hardware interrupt. For example, to attach an EDMA to external interrupt 4, use the EDMA channel 4.

For EDMA, before a transfer can be initiated, the parameters are loaded into the EDMA PRAM registers. This is performed by the `Submit()` method, which loads the PRAM with the transfer information. Unlike QDMA, this does not start the transfer itself. The transfer will be initiated when the associated hardware interrupt occurs. If using software triggering, use the `Set()` function to initiate a transfer. One `Set()` call is required for each link block in the transfer.

Each Edma transfer allocates blocks from the PRAM pool to configure its Link blocks. These blocks are a limited resource, and the allocation may fail. If the failure occurs, the `IsValid()` function will return false.

If a terminal count interrupt is not used, a call for `WaitForComplete()` will delay until the completion occurs. `TestComplete()` will return a flag that can be used to check completion without blocking.

Edma transfers may be configured to generate Terminal Count interrupts on completion of any and all blocks in the transfer. Which TC bit is signaled is configured in each settings block. This means there can be different handlers for different blocks in the transfer.

A user supplied handler, similar to an interrupt handler, can be associated with the terminal count interrupt by a call to the `TcIntInstall()` or `LinkTcIntInstall()` method. The Link function is used to install a handler for one of the link blocks as opposed to the primary block.

The DMA system shares a single interrupt for all TC interrupts, and the system will call the installed handler when the particular bit in the TC register becomes set. The handler installer requires an Interrupt Binder Object (See “Interrupt Binder Templates” on page 85.) as an argument to associate a handler function or method and argument for the interrupt forwarding mechanism of Pismo. A second pair of functions, `TcIntDeinstall()` and `LinkTcIntDeinstall()` removes any installed handler for the TC bit used by the block.

Once installed, TC interrupts for the entire transfer may be enabled or disabled by a call to `TcIntEnable()`.

The following example shows a full Edma transfer with TC interrupt handling. In this example a class member function is bound to handle the interrupt response.

```
class DmaIsr
{
public:
    typedef void (*IntFtnType)(void * fallow);

    DmaIsr()
        : Binder(*this, &DmaIsr::MyHandler, NULL)  {  }

    void MyHandler(void * fallow)
    {
```

```
            qdma_not_done = false;
        }

        ClassMemberHandler<DmaIsr, void *> Binder;
    };

    DmaIsr Isr;

    void EdmaTest()
    {
        Edma Ed;

        Ed.Settings().Priority(DmaSettings::priHigh).ElementSize(DmaSettings::is32bit);
        Ed.Settings().ElementIndex(1).ElementCount(50).FrameIndex(1).FrameCount(0);
        Ed.Settings().TCInt(true).TCCode(1).FrameSync(true);
        Ed.Settings().SourceAddr(int(&src_array[0])).SourceIncr(DmaSettings::Incr);
        Ed.Settings().DestinationAddr(dest_array).DestinationIncr(DmaSettings::Incr);
        //
        //  Define a linked
        DmaSettings Cfg;

        Cfg.Priority(1).ElementSize(0).SourceIncr(1).DestinationIncr(1);
        Cfg.TCInt(true).TCCode(1).FrameSync(true);
        Cfg.SourceAddr((int)&src_array[0]).DestinationAddr((int)(dest_array+50));
        Cfg.ElementCount(50).ElementIndex(1);
        Cfg.FrameCount(0).FrameIndex(1);

        Ed.AddLink(Cfg);
        Ed.LinkTcIntInstall( 0, Isr.Binder );
        Ed.TcIntClear();

        // This EDMA operation will trip a terminal count interrupt when
        // all data has been moved.
        InitArrays();
        Ed.TcIntEnable(true);
        qdma_not_done = true;
        Ed.Submit();
        // We software-initiate the EDMA here, but if this EDMA were using EINT4..7,
        // then an external int hardware pulse would remove need for Ed.Set, below
        Ed.Set();
        while (qdma_not_done)
            ;
        // Need to sync L2 cache with the of SDRAM, so that CPU can see the data
        CACHE_clean(CACHE_L2, dest_array, sizeof(dest_array));
        //
        //  Transfer the second transfer block...
        Ed.Set();
        while (qdma_not_done)
            ;
        // Need to sync L2 cache with the of SDRAM, so that CPU can see the data
        CACHE_clean(CACHE_L2, dest_array, sizeof(dest_array));
    }
```

The above example sets up a two block linked transfer triggered by software. A TC Interrupt is configured to signal the completion of each block in the transfer. The mainline waits for each block transfer to finish, as notified by the interrupt handler. Then the next block transfer is triggered by a second call to `Set()`. The Cache functions are required to assure that the cache and memory contents are back in synchronization.

Linked and Chained blocks

EDMA transfers may span multiple transfer blocks. On the completion of the primary transfer, the first link block is loaded into the primary block and initiated. When this block completes, the next linked block is loaded, and so on. A link block can

form a loop, but it is important to remember that the primary block can never be part of a loop. Since it is overwritten by the first linked transfer, this transfer can only occur once. Because of this to make a loop of two transfers requires **three** blocks to be configured. The primary block contains the first transfer, the first link the second transfer, and the third is a repeat of the first transfer that is linked back to the first link block.

Link blocks are allocated by a call to `AddLink()`. This call automatically configures the preceding block to link to this newly added block. It returns the index of the newly added block that can be used in order to configure the link block. To form a closed loop in a block chain, call `LinkBackTo()`. This connects the final block in the chain back to the block whose index is given in the argument.

Transfer chaining is a mechanism for having a transfer trigger another on completion. The `ChainTo()` and `ChainEnable()` methods set up a chaining relation between two transfers. Note that on the TI C671x processor, the second transfer must be configured on channels 8-11.

Class EdmaMaster
This class acts as a holder for functions and information common to all EDMA interrupts instead of associated with a single EDMA channel. Only one instance of EdmaMaster is created at program initialization. It is accessed by calling the static member function `EdmaMaster::Object()`.

EdmaMaster contains several functions dealing with the EDMA PRAM. This is a memory region shared among all EDMA objects giving a common storage for configuration blocks. This is a limited resource, so be wary of allocating many Edma blocks and not releasing them. The method `ClearPram()` clears all the PRAM blocks in a single operation.

EdmaMaster contains several functions dealing with the EDMA PRAM. This is a memory region shared among all EDMA objects giving a common storage for configuration blocks. This is a limited resource, so be wary of allocating many Edma blocks and not releasing them. Also available are functions to give access to the area at the end of the PRAM that is not used by the system. This scratchpad memory might be of use as a shared memory pool in an application.

Quadia and Duet Example Programs

Under **Quadia\Examples** in the install directory, the baseboard's example programs are installed. Some examples have no host component, and some use the terminal emulator applet as the host. Host examples are written in C++ either under Borland C++ Builder or Microsoft MSVC, or both. Target examples are written using CCS 3.x and DSP/BIOS.

Table 8. Quadia/Duet Example Programs

Example	Host	Target	Illustrates
CommonCDB	N/A	N/A	Shared CDB file for all examples.
CpuInRate	VC++7, VC8 (.NET)	DSP/BIOS	Shows data transfer from host machine in to the target CPU.
CpuOutRate	VC++7, VC8 (.NET)	DSP/BIOS	Shows data transfer from the target CPU out to the host machine.

Example	Host	Target	Illustrates
DuetLinkPort	BCB, BCB10, VC8 (.NET)	DSP/BIOS	Use of Link Ports for communication between processors on Duet.
Edma	terminal emulator	DSP/BIOS	Use of Pismo Edma and Qdma wrapper classes with installable interrupt handlers.
FftFix	terminal emulator	DSP/BIOS	Use of Fourier class to perform forward and inverse FFTs
Files	terminal emulator	DSP/BIOS	Use of C++ Standard I/O library
FirFix	terminal emulator	DSP/BIOS	Use of BlockFir class to perform FIR filter functions.
LinkPort	BCB, BCB10, VC8 (.NET)	DSP/BIOS	Use of Link Ports for communication between processors on the Quadia.
Swi	terminal emulator	DSP/BIOS	Use of Pismo SoftInt class for software interrupts.
Timer	terminal emulator	DSP/BIOS	Use of Pismo ClockBase objects for timebase control.

The Next Step: Developing Custom Code

In building custom code for an application, Innovative Integration recommends that you begin with one of the sample programs as an example and extend it to serve the exact needs of the particular job, or at least refer to the examples to see how some functions are done. Since each of the example programs illustrates a basic data acquisition or DSP task integrated into the target hardware, it should be fairly straightforward to find an example which roughly approximates the basic operation of the application. It is recommended that you familiarize yourself with the sample programs provided. The sample programs will provide a skeleton for the fully custom application, and ease a lot of the target integration work by providing hooks into the peripheral libraries and devices themselves.

Chapter 7. *Host/Target Communications*

Overview

Many applications involve communication with the host CPU in some manner. All applications at a minimum must be reset and downloaded from the host, even if they run independently from the host after that.

Other applications need to interact with a host program during the lifetime of the program. This may vary from a small amount of information to acquiring large amounts of data. Some examples:

- Passing parameters to the program at start time
- Receiving progress information and results from the application.
- Passing updated parameters during the run of the program, such as the frequency and amplitude of a wave to be produced on the target.
- Receiving alert information from the target.
- Receiving snapshots of data from the target.
- Sending a sample waveform to be generated to the target.
- Receiving full rate data.
- Sending data to be streamed at full rate.

These different requirements require different levels of support to efficiently accomplish. The simplest method supported is performing file I/O from within Code Composer using either the standard C file functions (which communicate directly through CCS to the Host file system), or via the Innovative terminal emulator, which supports simple data input and control and the sending of text strings to the user in addition to file I/O.

On the Velocia family baseboards, the CPU Busmaster interface allows communication of command messages and data packets between target and host. The command packets are fully interrupt driven and allow about 16 words of data to be transferred in each packet. For full rate data transfers, the hardware supports block-oriented bus-mastering transfers supporting maximum-speed data movement between the target and host.

CPU Busmastering Interface

Each TI C64x DSP on the baseboard is capable of independent and autonomous PCI bus-mastering to move data between target and host memory. This bus-master facility can be used to transfer data between host and target applications. In addition, the interrupts are also used to support the exchange of Message packets consisting of 16 words of data. This allows the exchange of command and parametric data without involving bus-mastering.

CPU Busmastering Implementation

Packet Based Transfers

Some Innovative DSP boards, such as the those within the Matador family, feature streaming bus-mastering hardware in which (logically) data is an infinite stream between the source and destination. This model is more efficient because the signaling between the two parties in the transfer can be kept to a minimum and transfers can be buffered for maximum throughput. On the other hand this streaming model has relatively high latency when attempting to communicate asynchronous data blocks, since a data item may stall in internal buffering until subsequent data accumulates to allow for an efficient transfer.

By contrast, the CPU bus-master interface implemented within the the C64x DSP transfers discrete blocks between the source and destination. Each data buffer is transferred completely to the destination in a single operation. Only if several transfers are requested at once will any delay in beginning transmission occur, as multiple requests have to be serialized through a single hardware system.

The data buffers transferred can be of different sizes. Each requested buffer is interrogated for its size and fully transmitted. At the destination, the destination buffer is re-sized to allow the incoming data to fit. If the buffer given is too small for the data, it will be reallocated to allow the transfer. Reallocating buffers can take some time, for best performance buffers should be presized to be large enough for the largest transfer expected. This will make allocation of buffers at critical times unnecessary.

Blocking Interface

CPU bus-mastering uses a simple blocking interface for its send and receiving functions. The sending function will not return until the transfer has completed and the buffer is ready for reuse. Similarly, the receiving function waits until data has arrived from the data source and transferred into the data buffer before returning. At this point the buffer is ready for use. This blocking allows sequences of transfers managed by a simple sequence of calls to transfer functions.

Since the transfer functions are blocking, they are best avoided in the main user interface thread of a Windows application. The GUI will be appear to be ‘frozen’ until the transfer has completed. For best results, the data transfer functions should be placed in separate threads on the target and host applications. In fact, each direction of transfer should have its own thread, so that the two directions of transfer can interleave as much as possible. The example programs CpuBmIn and CpuBmOut illustrate the use of separate threads for data transfer.

Maximum Transfer Size

The largest transfer allowed is half of the total size of the DMA Buffer allocated by the INF file when the driver is installed. Half of the memory is dedicated to each direction. The default buffer size in the INF is 0x200000 bytes, so the maximum transfer is 1 Megabyte.

Malibu Library Host Support for CPU Busmastering

In concept, there are a large number of ways that data can flow data between PCI resources. Data can be be bus-mastered or slave accesses can be used. When bus-mastering, data can flow continuously, referred to as streaming, or intermittently. Consequently, Malibu has been designed such that baseboard objects such are Quadia or C64xDsp do not contain embedded support for data flow via the PCI bus. Rather, in order to isolate encapsulate the details of particular bus-mastering strategies and provide a means by which baseboards can perform the type of data flow most appropriate for each application, baseboard objects must be logically connected to an independent communications object which is responsible for communicating data to and from Host memory. Logically, a baseboard is *connected* to a communications object which implements a particular communications strategy.

The TiBusmasterStream object is one such communications object. Applications instantiate an object of this type, then associate it with a baseboard object in order to allow it to perform communications functions. The TiBusmasterStream::ConnectTo() method is used to establish this association. Once connected in this fashion, the Send() and Recv() methods may be used to transfer buffers of data between the Host CPU and DSP baseboard.

```
//
//  Block Transfer System Methods
virtual bool  Send( const IntegerBuffer & packet );
virtual bool  Recv( IntegerBuffer & packet );
```

`TiBusmasterStream::Send()` sends the contents of a `IntegerBuffer` object to the target. All of the data in the buffer is transferred. There is no means of sending a partial buffer. The function will not return until the entire block has been transferred to the recipient DSP.

The function returns true if the transfer succeeded. It returns false if the transfer failed due to a PCI bus error.

`TiBusmasterStream::Recv()` waits for data to arrive from the target, then returns the data in the buffer provided. The `IntegerBuffer` buffer will automatically be re-sized to fit the data transferred from the source. If the buffer is smaller than the amount of data received, this may involve a reallocation of the data block.

The function returns true if the transfer succeeded. It returns false if the transfer failed due to a PCI bus error.

Packet Notification Events

The TiBusmasterStream object contains an event that will be signaled when a packet buffer arrives from the target. This OnPacketAvailable event can have a handler installed that will process the message, thus eliminating the need for a separate thread to manage incoming data packets. The Recv() method can be called in the handler with the assurance that the request will not block, since data is already present.

Target (Pismo Library) Support for CPU Busmastering

In the Pismo library, the UtilLib library contains a file PciTransfer.h that contains this class:

```
class PciTransfer : public PciTransferBase
{
public:
        PciTransfer();

        bool  Send(int channel, const Buffer & buffer);
        bool  Recv(int channel, Buffer & Buffer);
```

```
        ...
};
```

`PciTransfer::Send()` sends the contents of a Buffer-derived object to the Host. All of the data in the buffer is transferred. There is no means of sending a partial buffer. The function will not return until the block has been transferred to the host.

The use of the base buffer class allows any of the `IntBuffer, CharBuffer, FloatBuffer` and similar classes to be sent across the interface.

The function returns true if the transfer succeeded. It returns false if the transfer failed due to a PCI bus error.

`PciTransfer::Recv()` waits for data to arrive from the target, then returns the data in the buffer provided. The Buffer will be re-sized to fit the data transferred from the source. If the buffer is too small, this may involve a reallocation of the data block.

The function returns true if the transfer succeeded. It returns false if the transfer failed due to a PCI bus error.

Packetized Message Interface

The C64x processor's PCI interface is also used to support a lower bandwidth communication link for sending commands or parametric information between target and host. Library support is provided to build a packet-based message system between the target and host software. These packets can provide a simple yet powerful means of sending commands and information across the link between the two processes.

Message Mailbox Emulation

On Matador baseboards, a set of sixteen mailboxes in each direction to and from the host PC are shared with the DSP to allow for an efficient message mechanism that complements the bus-mastering interface. These mailboxes have a handshake mechanism that signals the recipient for the availability of data, and a corresponding signaling to the sender when the message was received.

The C64x processor has no such facility, but the software has been designed to emulate the same interface. Message Packets of 16 words are cached in a memory region that can be accessed by the host when signaled that data is available. Interrupts are used for signaling in both directions, allowing for rapid response. But since the packets themselves are small, throughput is relatively modest. The packet interface with bus mastering is preferred for large volumes of data.

Software in the Pismo Toolset and Malibu implement a message system that allows the application programmer to use the mailbox system for command and control, lower rate data passing, and status queries and replies, as well as many other uses.

The Message System

The Message system provides a single bi-directional link between the target and the host. More complicated arrangements are possible by using the messages to encode data sources and destinations. Two words of the message are dedicated to this kind

of routing information, with three fields defined for routing or message type encoding. The remaining space can be used for data.

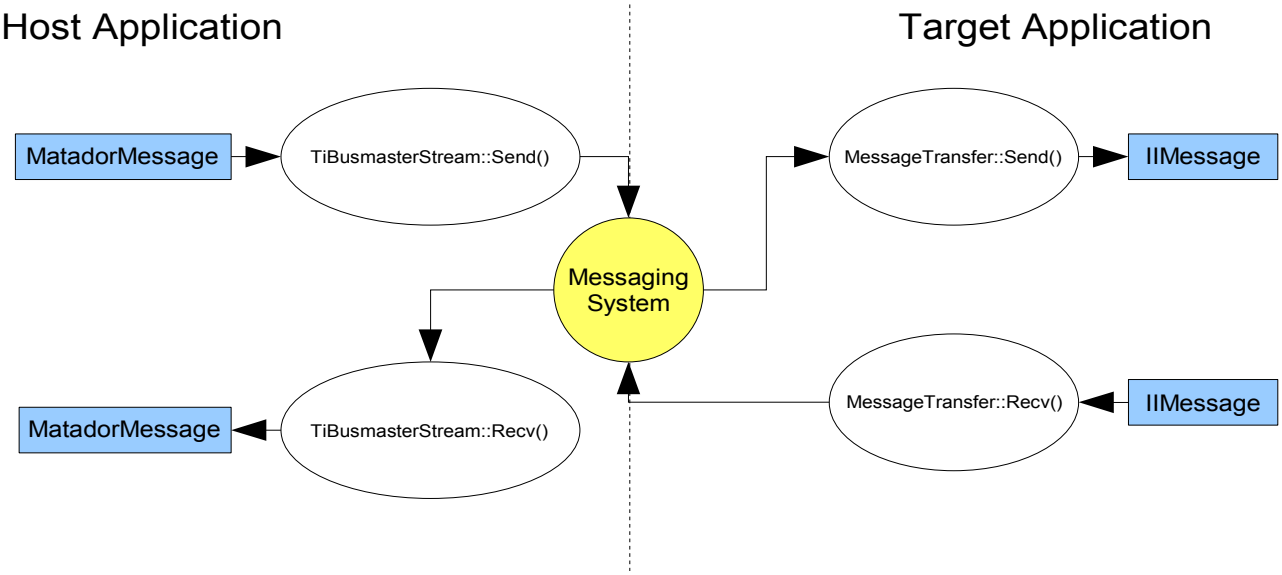


Figure 20. Messaging System Objects

Host-side Message Objects

Messages consist of packets that may contain up to 14, 32-bit data words plus two 32-bit header words. The details of packet formatting are hidden on both the target and the host by the use of similar Message objects to encapsulate the packet to be transmitted. On the host side this message packet class is called `MatadorMessage`. On the target, the corresponding class is called `IIMessage`.

Messages sent by the target are collected into `MatadorMessage` objects for delivery to the event handlers dedicated to responding to the messages. For all practical purposes, you can think of the Message System as exchanging `IIMessage/MatadorMessage` objects.

The header portion of the Message Packet contains some system data and some fields that can be used by the application. The `TIIMessage` Object provides properties to access these fields:

Table 9. TIIMessage Header Field Access

Channel	Property	Message Channel (Free for use in application).
TypeCode	Property	Message or Command Type
MessageId	Property	Message counter or other user data
IsReplyExpected	Property	Set if reply is needed. Free for use in application.

The 14 words of Data are accessible as arrays. Array properties are defined to allow loading common data types into a message:

Table 10. TIIMessage Data Section Interface

Data[]	Property	Access the data region as 32-bit integers (0-13)
AsFloat[]	Property	Access the data region as floating point data. (0-13)
AsShort[]	Property	Access the data region as 16-bit integers. (0-27)
AsChar[]	Property	Access the data region as 8-bit characters. (0-55)

Message Packets supporting a mix of data formats are supported as long as the user remembers to make sure the individual portions do not collide in the message data. For example when mixing a **float**, a **char** and a **short** and an **int** (in that order) the index of the float is 0, the char is at char-index 4, the short is at short-index 4, and the integer can go at integer index 2. Creating a wrapper class to handle the indexes can make the use of mixed-mode indexes transparent to the user. Failing that, using only 32 bit wide data for floats and integer types makes the indexing clear, at the expense of data packing efficiency.

Target Side Message Objects

On the target side, the Pismo library supports a very similar class, `IIMessage`, to contain the message. However, since on the target Properties are not supported, we instead use a convention to define setter and getter functions in a uniform manner. These paired methods are called "a property" or "property methods". (See the Pismo Online Help under 'property' for more details). The convention is to overload the method name, distinguishing getter function by a 'const' declaration and the setter by an additional argument for the value to set. For example, consider the abridged definition of the ClockBase class below:

```
class ClockBase : public CslNoncopyable
{
public:

    ...

    // Enabled is a setter/getter property
    void            Enabled(bool state);
    bool            Enabled() const;

    ...
};
```

The following table gives the header field access methods for the class::

Table 11. IIMessage Header Field Access

Channel()	Property Methods	Message Channel (Free for use in application)
TypeCode()	Property Methods	Message or Command Type
MessageId()	Property Methods	Message counter or other user data
IsReplyExpected()	Property Methods	Set if reply is needed. Free for use in application.

The 14 words of Data are accessible as array property methods. These methods all have an additional argument giving the index into the data section:

Table 12. IIMessage Data Section Interface

Data()	Property Methods	Access the data region as 32-bit integers (0-13)
AsFloat()	Property Methods	Access the data region as floating point data. (0-13)
AsShort()	Property Methods	Access the data region as 16-bit integers. (0-27)
AsChar()	Property Methods	Access the data region as 8-bit characters. (0-55)

Message Packets supporting a mix of data formats are supported just as on the host side.

Message Communication

The packetized message system is event driven. When the sender posts a message packet, at the first available opportunity the packet is loaded into a reserved memory region and an interrupt is generated to the receiving side. On the receiver, the interrupt is detected and the application thread waiting for the messages is notified. After processing the message, the sender receives an acknowledgment that the previous packet has been processed and the memory region is free for another transmission. The application at the same time processes the message data and performs whatever action is needed.

To establish a bi-directional link you need a sender and a receiver on both the target and the host.

The 14 words of `Data` are accessible as array property methods. These methods all have an additional argument giving the index into the data section:

Table 13. Message Sending and Receiving Methods

Direction	Sender	Sender Type	Receiver	Receiver Type
Target to Host	MessageTransfer::Send()	Member Function	TiBusmasterStream::Recv()	Blocking call
Host to Target	TiBusmasterStream::Send( )	Member Function	MessageTransfer::Recv()	Blocking call

Before using messages, the target and the host must instantiate threads which are responsible for receiving and sending all message traffic. Within these threads, the user code will block in any call to Send or Recv until the message has been communicated from the sender to the receiver. Consequently, these calls must not be made from within the main thread within Windows applications, since that thread must respond to system messages.

On the host side, you can use events to provide notification of the arrival of a message. If you provide a handler for the OnMailAvailable event in the TiBusmasterStream object, the handler will be called on the arrival of a new mail message. This notification can be synchronized with the main GUI thread to allow updates to the user interface. Since the message has already arrived, calling Recv() inside the event handler will not cause the main GUI thread to block.

The messaging system uses the same interrupt subsystem as does the bus mastering interface. Therefore, messages may be transferred in either direction while data streaming is in progress, but bus-mastering transfers and messages will be serialized. Under normal circumstances, there may be some minimal bus-mastering speed degradation due to the increased load, but given that the messaging system is only designed for moderate bandwidth communication this should not be significant. Since the signaling of data available and the acknowledgment require interrupts, it is a requirement that global interrupts be enabled for messages to proceed.

C++ Terminal I/O

The terminal emulator applet is a Host PC application which provides a C++ language-compatible, terminal emulation facility for interacting with the TermIo Pismo library running on an Innovative Integration DSP processor.

Using the terminal emulator, it is possible to develop and debug target DSP code while deferring development in Host application code. By using simple, streaming I/O functions within a target application during development, DSP algorithms can be developed independently from Host applications. Later, when a custom Host application code is written, the DSP standard I/O functions may be deleted from the target application and the target application will no longer be dependent on the emulator or the target TermIo libraries.

Streaming methods such as << and >> are dispatched by the TermIo object to route text and data between the DSP target and the Host terminal terminal emulator applet. Text strings are presented to the user via a terminal emulation window and host key-board input data is transmitted back to the DSP. The terminal emulator works almost identically to console-mode terminals common in DOS and Unix systems and provides an excellent means of accessing target program data or providing a simple user interface to control target application operation.

Target Software

All of the features of the terminal are accessed through the two classes TermIo and TermFile. TermIo provides the basic streaming interface which allows text messages to be formatted and streamed out to the terminal as well streaming in strings and numeric values from the terminal for consumption by target application code. The TermFile class provides a mechanism allowing target applications to open host disk files, perform read and write accesses, and subsequently close these files. See the Files.cpp example for illustrative usage of each of these classes and their functions.

Tutorial

Using the terminal during target software development is simple. The global cio object which is automatically instantiated within the Pismo libraries. Use the methods within the cio class to format text strings and then stream them to the Terminal applet

```
cio << bold << "7Demonstrate file I/O\n\n" << normal << endl;
```

Note the use of manipulators, such as bold and normal, to force formatting of the text string as it is streamed to the host. TermIo features many such manipulators to perform functions such as setting text color (setcolor), clearing to end-of-line (clreol), clearing the screen (cls) and so forth. Other manipulators are available to format numeric values as they are streamed to the host. For example, the phrase

```
cio << "Hello" << hex << showbase << 4660 << dec;
```

displays the string "Hello 0x1234" on the console display, converting the integer value 4660 as a hexadecimal number on the target, prior to streaming it to the host. Other manipulators are available providing extensive control over the display of floating point numbers as well as integer values.

It is also frequently necessary to obtain input from an operator during the run-time execution of a target application. For example, it may be necessary to prompt for a sample rate at which analog I/O is to be streamed. The code fragment below illustrates the necessary technique

```
// Prompt the user
cio << " Enter a float: " << flush;
float x2;
// Eat user input
cio >> x2;
```

The stream manipulator >> is overloaded to allow streaming directly into floating point, integer and string variables directly from UniTerminal.

To perform file input and output from within target applications, first instantiate a TermFile object, as below

```
TermFile File;
```

Then, use the TermFile Open method to open the file for access on the host using the desired open attributes

```
if (!File.Open("wave.bin", "w+b"))
    {
    cio << "nOutput file open error - Program terminating!" << endl;
    cio.monitor();
    }
```

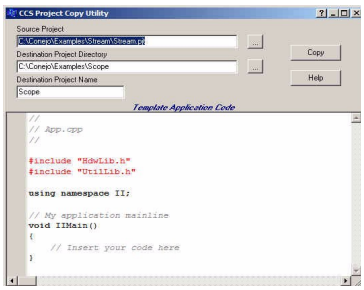
This method returns a Boolean indicating success if the file open is successful. To store data into the file or retrieve data from the file, use the Write or Read methods, respectively. For example

```
transferred = File.Write((char*)&Buffer[0], 10000);
```

writes 10000 bytes of Buffer into the disk file. When disk operations have been completed, the file should be closed using the `TermFile::Close` method.

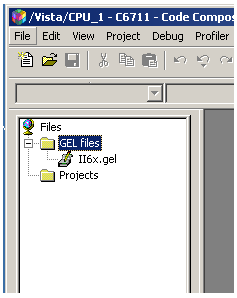
Chapter 8. *Building a Target DSP Project*

Building a project suitable for a Matador or Velocia baseboard requires a particular setup of the project.

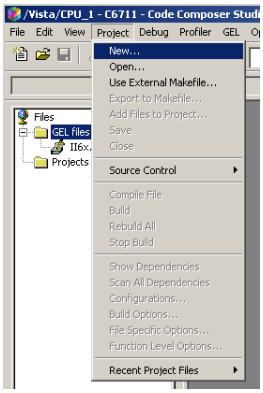


By far, the easiest way to create a new DSP project is by using an existing project as a template. The CopyCcsProject applet provided in the Pismo Toolset automates this task. To use this utility, select an existing Code Composer project as the Source Project, typically one of the example programs supplied in the Pismo Toolset. Next, select the directory into which you wish the new project to be created, using the Destination Project Directory edit control. Then, edit the Destination Project Name for the newly-created project. Finally, click the Copy button to create the new project from the template. The new project may be opened and used within Code Composer.

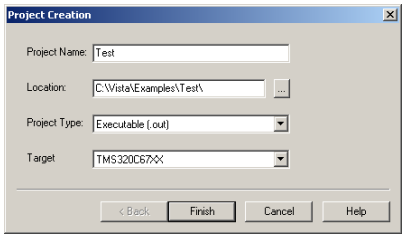
Alternately, you may follow the manual steps below to create a new target DSP project. The project name used below is called Test, but you should name your project appropriately for your application.



Start Code Composer Studio. In the default configuration, the project window will contain no projects but will contain the default Innovative-supplied board initialization GEL file.

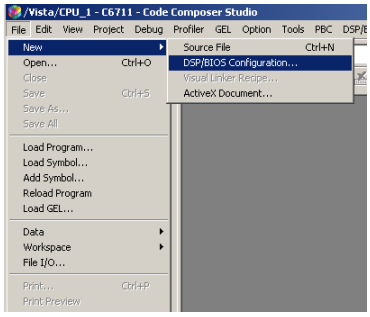


Click Project | New on the menu bar to create a new DSP project.

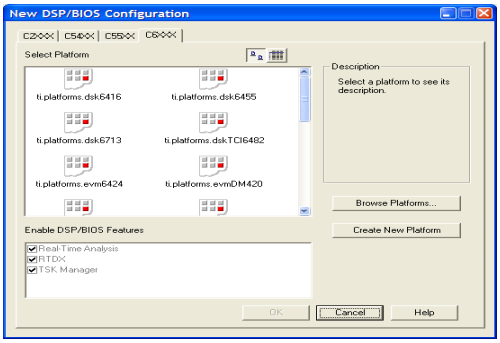


Specify the location for the new project and its name. In this example, a new project called Test is being created in the `Sbc6711-Pismo\\Examples\\` directory. Change the location to accommodate your board type and processor type.

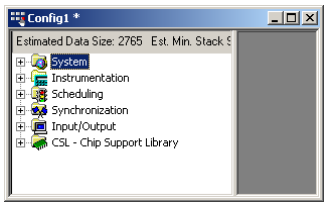
After the new project has been created, it will appear in the CCS project window under the Projects folder.



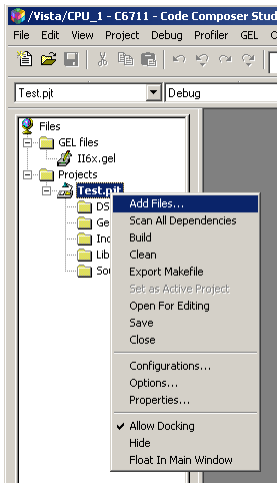
Click File | New | DSP/BIOS Configuration to create a new TCF file for use in the project.



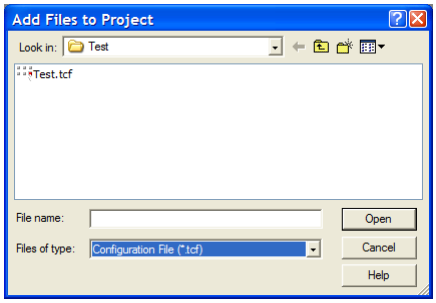
Select the relevant template for the baseboard from the list in the New DSP/BIOS Configuration dialog box.



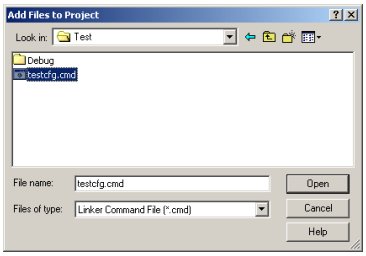
By default, this TCF will be named Configuration1. Save it as `Test.TCF`.



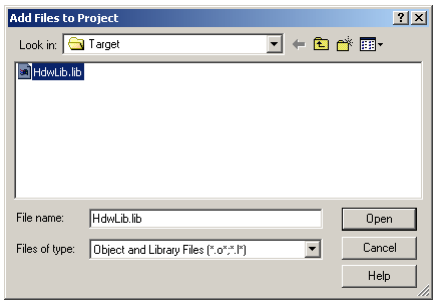
Though the TCF and its support files have been created on disk, you must manually add them to the Test project. Right-click on `Test.pjt` in the Project window to invoke the project hot menu. Click Add Files to add a file to the project.



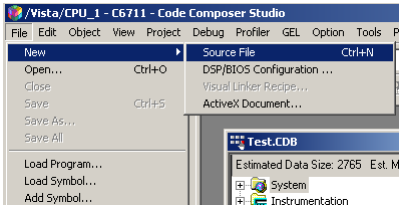
Select the the newly-created `Test.tcf` for addition to the project. This will implicitly add the auto-generated files `Testcfg.s62 (Testcfg.s64 for Velocia cards)` and `Testcfg_c.c` to the project as well.



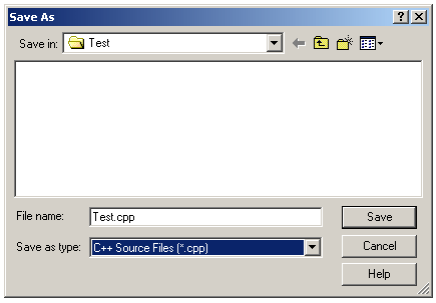
Right-click on `Test.pjt` in the project window, click “Add Files” then select the the newly-created `Test.cmd` for addition to the project.



Right-click on `Test.pjt` in the project window, select Add Files, then browse to the Examples directory and select `Examples.cmd` for addition to the project.



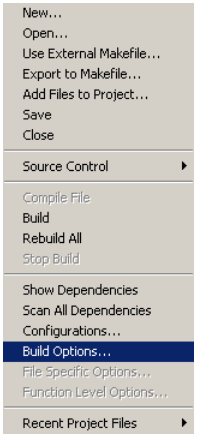
Add an new C++ source file to the project. Click File | New | Source File to create an empty source document.

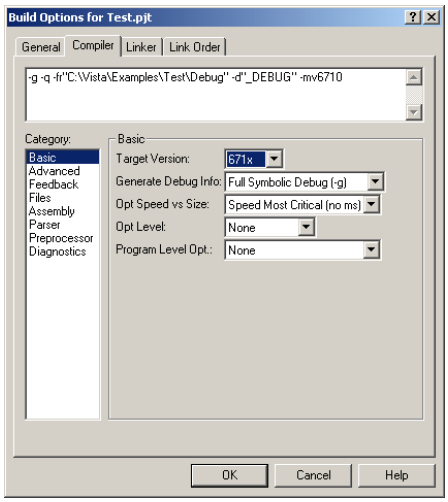


Rename the new source document to `Test.cpp`. To use the Pismo libraries, you must use C++ files and the C++ compiler, even if you intend to restrict your own coding to the C subset of C++

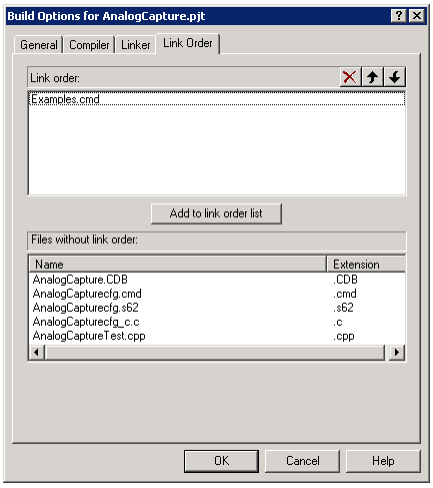
```
//
// App.cpp
//

#include "Pismo.h"

using namespace II;

// Application mainline
void IIMain()
{
    // Insert your code here
}
```

Type the boilerplate code below into your source file. This is the minimum code needed for any Pismo C++ application.

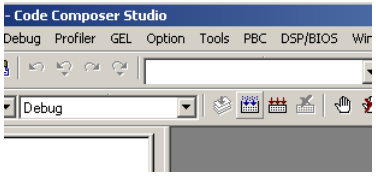




Click the menu Project | Build Options to invoke the compiler Build Options dialog. Then select the Files Category, then enter the pathspec to the `Examples.opt` file in the Examples directory to the Options File edit box.



Click on the Link Order tab, then add `Examples.cmd` to the Link Order List.



Click the Incremental Build button to rebuild the template application. It should compile and link without errors.

Writing a Program

The basic program given in the example above includes a 'Main' function `IIMain()`. DSP/BIOS, the OS used in the Pismo library, uses code inserted after exiting from the normal C-language `main()` to initialize features of DSP/BIOS. This means that some language features are not available then. To avoid these problems the Pismo library provides a `main()` function and uses it to create a single thread. This thread, when executed, calls the `IIMain()` function. Inside of this thread, all DSP/BIOS is initialized and ready for use. It is required that the user include this function, and use it as the equivalent of the old main process function in C.

Host Tools for Target Application Development

The Innovative Integration Pismo Toolset allows users of Innovative DSP processor boards to develop complete executable applications suitable for use on the target platform. The environment suite consists of the TI Optimizing C++ Compiler, Assembler and Linker, the Code Composer debugger and code authoring environment as well as Innovative's custom Windows applets (such as the terminal emulator).

Code Composer Studio is the package used to automate executable build operations within Innovative's Pismo Toolsets, simplifying the edit-compile-test cycle. Source is edited, compiled, and built within Code Composer Studio, then downloaded to the target and tested within either the Code Composer Studio debugger or via the terminal emulator.

Code Composer Studio may be used for both code authoring and code debugging. Details of constructing projects for use on Innovative DSP platforms are given in the above section of this chapter.

Do not confuse the creation of target applications (code running on the target DSP processor) with the creation of host applications (code running on the host platform). The TI tools generate code for the TI DSP processors, and are a separate toolset from that needed to create applications for the host platform (which would consist of some native compiler for the host processor, such as Microsoft's Visual C++ or Borland Builder C++ for IBM compatibles). To create a completely turnkey application with custom target and host software, *two* programs must be written for two separate compilers. While Innovative supports the use of Microsoft C/C++ for generation of host applications under Windows with sample applications and libraries, we do not supply the host tools as part of the Development Environment. For more information on creating host applications, see the section in this manual on host code development.

This section supplies information on the use of the development environment in creating custom or semicustom target DSP software. It is not intended as a primer on the C++ language. For information on C/C++ language basics, consult one of the primer books available at your local bookstore.

Components of Target Code (.cpp, .tcf, .cmd, .pjt)

In general, DSP applications written in TI C++ require at least three files: a .cpp file (or "source" file) containing the C++ source code for the application a .cmd file (or "command" file) which contains the target-specific memory-map and build data needed by the linker, a .tcf file (or "command database" file) which specifies the properties of the BIOS operating system used within the application and a .pjt file ("project" file) which centralizes all project-specific options, settings and files. There may also be one or more .asm assembler source files, if the user has coded any portions of the application in assembly language.

Edit-Compile-Test Cycle using Code Composer Studio

Nearly every computer programming effort can be broken down into a three step cycle commonly known as the edit-compile-test cycle. Each iteration of the cycle involves editing the source (either to create the original code or modify existing code), followed by compiling (which compiles the source and creates, or builds, the executable object file), and finally downloading and testing the result to see if it functions in the desired fashion. In the Innovative Integration development system these stages are accomplished within the Code Composer integrated development environment (IDE).

By using Code Composer Studio, these stages of the programming cycle are accomplished entirely within the IDE. The project features of Code Composer Studio support component file editing and compilation stages, along with allowing the executable result to be downloaded and tested on the target hardware. This fully integrated programmers environment is more user-friendly then the basic command line interface, which comes standard with the TI tools.

Automatic projectfile creation

When a project is created, opened, modified, built or rebuilt, the Code Composer Studio dependency generator automatically generates a project makefile (named `<project file>.pjt`, located in the project directory), which is capable of rebuilding the project's output file from its components.

This file is automatically submitted to the internal make facility whenever you click on build or rebuild within Code Composer Studio. The make facility automatically constructs the output file by recompiling the out-of-date source files including the dependencies contained within those source files.

Rebuilding a Project

It is sometimes necessary to force a complete rebuild of an output file manually, such as when you change optimization levels within a project. To force a project rebuild, select Project | Rebuild All from the Code Composer Studio menu bar.

IIMain replaces main.
Due to restrictions within Dsp/Bios, not all BIOS features may be safely used within `main()`, since it is called early in the system initialization sequence. To circumvent this limitation, Pismo automatically constructs a default thread running within normal priority and starts this thread automatically. The entry point function in this thread is called IIMain, and all Pismo applications must define this function. This function is intended to replace main in your application programs. You may safely call any BIOS function within IIMain.

Running the Target Executable

The test program may be converted into a simple, "Hello World!" example, by using the built-in standard I/O features within Pismo. Bring up the `Test.cpp` source file edit screen. Scroll down the source file by using cursor down button until you reach the `IIMain()` function. Edit it as follows:

```
#include "HdwLib.h"
#include "UtilLib.h"

        cio << init;
cio << "Hello World!" << endl;
```

```
cio.monitor();
```

You can now compile the new version by executing Build from the Project menu (or by clicking on its toolbar icon). This causes Code Composer Studio to start the compiler, which produces an assembly language output. The compiler then automatically starts the assembler, which produces a `.obj` output file (`test.obj`). Code Composer Studio then invokes the TI Linker using the `testcfg.cmd` file, which is located in the project directory. This rebuilds the executable file using the newly revised `test.obj` . If no errors were encountered, this process creates the downloadable COFF file `test.out`, which can be run on the target board. At this point, the program may be run using the terminal emulator applet, which may be invoked using the terminal emulator shortcut located within the target board program group created during the Pismo Libraries installation process. In the terminal emulator, download the `test.out` file. The program runs and outputs the message "Hello, World" to the terminal emulator window.

If errors are encountered in the process, Code Composer Studio detects them and places them in the build output window. If the error occurred in the compiler or assembler (such as a C++ syntax error), the cursor may be moved to the offending line by simply double-clicking on the error line within the build output window, and the error message will be displayed in the Code Composer Studio status bar. If the linker returns a build error, the build output window shows the error file. From this information, the linker failure can be determined and corrected. For example, if a function name in a call is misspelled, the linker will fail to resolve the reference during link time and will error out. This error will be displayed on the screen in the build output window.

Note:
Be sure to start the terminal emulator **BEFORE** starting Code Composer, to avoid resetting the DSP target in the midst of the debugging session. If the terminal emulator is not yet running and you wish to run the Test object file, perform the following steps.

1. Execute Debug | Run Free to logically disconnect the DSP from the debugger software.

2. Terminate the Code Composer Studio application.

3. Invoke the terminal emulator application.

4. Restart the Code Composer Studio application.

This outlines the basics of how to recompile the existing sample programs within the Code Composer Studio environment.

Anatomy of a Target Program

While not providing much in the way of functionality, the `test` program does demonstrate the code sequence necessary to properly initialization the target. The exact coding, however, is very specific to the I.I. C Development Environment, target boards, and is explained in this section in order to acquaint developers with the basic syntax of a typical application program.

```
/*
*       HELLO.CPP
*       Test file/program for target board.
*/

#include "Pismo.h"
```

```
IIMain()
{
        cio << init;

        cio << "Hello World!" << endl;
        cio << "\nEchoing keystrokes..." << endl;

        char key;

        do
                {
                cio >> key;
                cio << key << flush;
                }
        while(key != 0x1b);

        cio.monitor();
}
```

The two lines of the program that being with a "#" are #include statements, which include the header files for the hardware and utility I/O libraries. These include prototypes for all the library classes within Pismo.

The `cio << init` invocation will setup the standard monitor I/O interface and reset the terminal window. The next lines perform the basic standard I/O functions of printing "Hello World!" & "Echoing keystrokes...". These two lines are where custom code could be inserted.

The following do-loop sequence simply echoes keys typed at the terminal emulator back to the terminal display, until the `Esc` key is pressed. When `Esc` is pressed, the `cio.monitor()` function effectively terminates the program, except that interrupts are still active and interrupt handlers (if they had been installed) would still execute properly.

The `test` program is very simple, but it contains the basic components of a typical DSP application, as well as the initialization needed to interact with the terminal emulator.

Use of Library Code

Library routines can be compiled and linked into your custom software simply by making the appropriate call in the source and adding the appropriate library to the linker command file. Refer to the library reference within the Pismo online help for library location information on each class and method.

In general, user software needs to #include the relevant library header file in source code. The header files define prototypes for all library functions as well as definitions for various data structures used by the library functions. The files `HdwLib.h` and `UtilLib.h` should be included within all programs; The file `DspLib.h` should be included if a program uses functions in the DspLib signal processing library.

Example Programs

Under **<baseboard>\Examples** in the install directory, the baseboard's example programs are installed. Some examples have no host component, and some use the terminal emulator applet as the host. Host examples are written in C++ either under

Borland Builder or Microsoft MSVC, or both. Target examples are written using CCS 3.3 and DSP/BIOS. Note that not all of the examples listed below are available for all targets.

Table 14. Pismo Example Programs

Example	Host	Target	Illustrates
FftFix	terminal emulator	DSP/BIOS	Use of Fourier class to perform forward and inverse FFTs
FirFix	terminal emulator	DSP/BIOS	Use of BlockFir class to perform FIR filter functions.
Edma	terminal emulator	DSP/BIOS	Use of Pismo Edma and Qdma wrapper classes with installable interrupt handlers.
Files	terminal emulator	DSP/BIOS	Use of C++ Standard I/O library
CpuInRate	BCB MSVC	DSP/BIOS	Use of Target to Host message and data packet passing via PCI bus.
CpuOutRate	BCB MSVC	DSP/BIOS	Use of Host to Target message and data packet passing via PCI bus.
LinkPort	BCB	DSP/BIOS	Use of LinkPort driver to flow data between all processor in mesh
Swi	terminal emulator	DSP/BIOS	Use of Pismo SoftInt class for software interrupts.
Timer	terminal emulator	DSP/BIOS	Use of Pismo ClockBase objects for timebase control.

The Next Step: Developing Custom Code

In building custom code for an application, Innovative Integration recommends that you begin with one of the sample programs as an example and extend it to serve the exact needs of the particular job. Since each of the example programs illustrates a basic data acquisition or DSP task integrated into the target hardware, it should be fairly straightforward to find an example which roughly approximates the basic operation of the application. It is recommended that you familiarize yourself with the sample programs provided. The sample programs will provide a skeleton for the fully custom application, and ease a lot of the target integration work by providing hooks into the peripheral libraries and devices themselves.

Chapter 9. *Developing Host Applications*

Developing an application will more than likely involve using an integrated development environment (IDE) , also known as an integrated design environment or an integrated debugging environment. This is a type of computer software that assists computer programmers in developing software.

The following sections will aid in the initial set-up of these applications in describing what needs to be set in Project Options or Project Properties.

Borland Turbo C++

BCB10 (Borland Turbo C++) Project Settings

When creating a new application with File, New, VCL Forms Application - C++ Builder

Change the Project Options for the Compiler:

Project Options
++ Compiler (bcc32)
C++ Compatibility
Check 'zero-length empty base class (-Ve)'
Check 'zero-length empty class member functions (-Vx)'

In our example Host Applications, if not checked an access violation will occur when attempting to enter any event function.

i.e.
Access Violation OnLoadMsg.Execute – Load Message Event
Because of statement
Board->OnLoadMsg.SetEvent(this, &ApplicationIo::DoLoadMsg);

Change the Project Options for the Linker:

Project Options
Linker (ilink32)
Linking – **uncheck 'Use Dynamic RTL'**

In our example Host Applications, if not unchecked, this will cause the execution to fail before the Form is constructed.

Error: First chance exception at $xxxxxxxx. Exception class EAccessViolation with message "Access Violation!"
Process ???.exe (nnnn)

Other considerations:

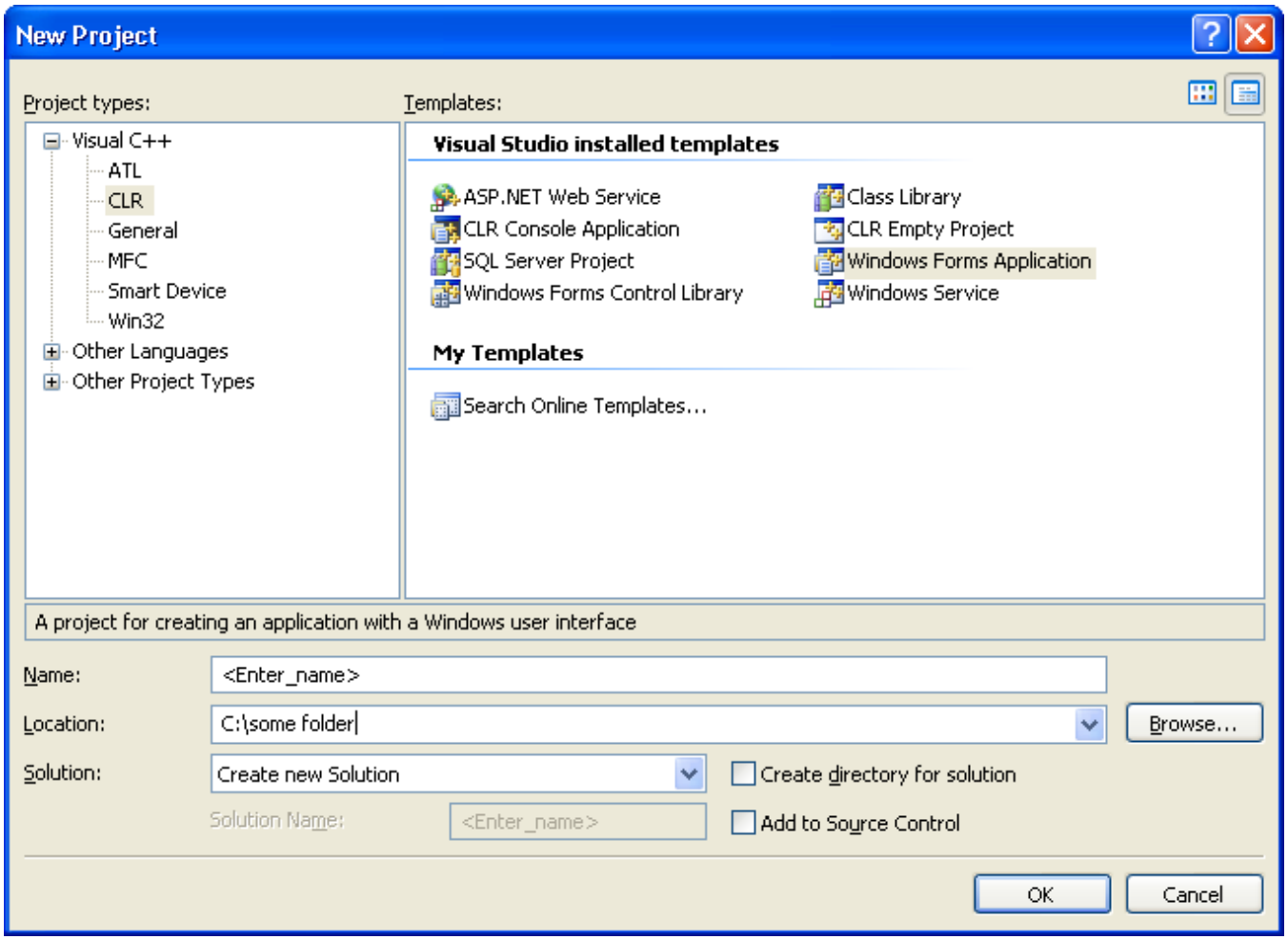
Project Options
++ Compiler (bcc32)
Output Settings
check – Specify output directory for object files(-n)
(release build) Release
(debug build) Debug
Paths and Defines
add Malibu
Pre-compiled headers
uncheck everything
Linker (ilink32)
Output Settings
check – Final output directory
(release build) Release
(debug build) Debug
Paths and Defines
(ensure that Build Configuration is set to All Configurations)
add Lib/Bcb10
(change Build Configuration to Release Build)
add lib\bcb10\release
(change Build Configuration to Debug Build)
add lib\bcb10\debug
(change Build Configuration back to All Configurations)

Packages
uncheck - Build with runtime packages

Microsoft Visual Studio 2005

Microsoft Visual C++ 2005 (version 8) Project Properties

When creating a new application with File, New, Project with Widows Forms Application:



Project Properties (Alt+F7)
Configuration Properties

Project Defaults	
Configuration Type	Application (.exe)
Use of MFC	Use Standard Windows Libraries
Use of ATL	Not Using ATL
Minimize CRT Use in ATL	No
Character Set	Use Unicode Character Set
Common Language Runtime support	Common Language Runtime Support (/clr)
Whole Program Optimization	No Whole Program Optimization

C++
General
Additional Include Directories

Malibu
PlotLab/Include – for graph/scope display

Code Generation
Run Time Library

Multi-threaded Debug DLL (/Mdd)

Precompiled Headers
Create/Use Precompile Headers

Not Using Precompiled Headers

Linker
Additional Library Directories

Innovative\Lib\Vc8

If anything appears to be missing, view any of the example sample code Vc8 projects.

DialogBlocks

DialogBLocks Project Settings (under Linux)

Project Options
[Configurations]
Compiler name = GCC
Build mode = Debug
Unicode mode = ANSI
Shared mode = Static
Modularity = Modular
GUI mode = GUI
Toolkit = <your choice wxX11, wxGTK+2, etc>
Runtime linking = Static or Dynamic, we use Static to facilitate execution of programs out of the box.
Use exceptions = Yes
Use ODBC = No
Use OpenGL = No
Use wx-config = Yes
Use insalled wxWidgets = Yes
Enable universal binaries = No
...
Debug flags = -ggdb -DLINUX
Library path = %INNOVATIVE%/Lib/Gcc/Debug, %WINDRIVER%/lib
Linker flags = %AUTO% -Wl, @%PROJECTDIR%/Example.lcf
IncludePath= -I%INNOVATIVE%/Malibu -I%INNOVATIVE%/Malibu/LinuxSupport %AUTO%

[Paths]
INNOVATIVE= /usr/Innovative
WINDRIVER= /usr/Innovative/WinDriver
WXWIN= /usr/wxWidgets-2.8-7 provided that this is the location where you have installed wxWidgets.

Summary

Developing Host and target applications utilizing Innovative DSP products is straightforward when armed with the appropriate development tools and information.

Chapter 10. *Applets*

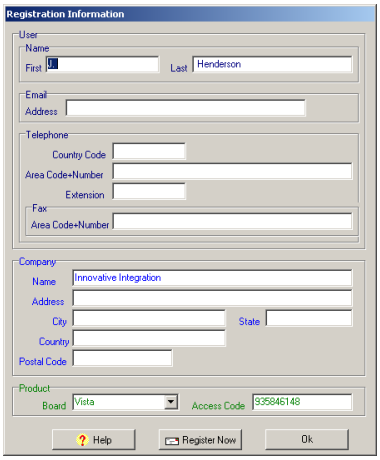
The software release for a baseboard contains programs in addition to the example projects. These are collectively called "applets". They provide a variety of services ranging from post analysis of acquired data to loading programs and logic to a full replacement host user interface. The applets provided with this release are described in this chapter.

Shortcuts to these utilities are installed in Windows by the installation. To invoke any of these utilities, go to the Start Menu | Programs | <<Baseboard Name>> and double-click the shortcut for the program you are interested in running.

Common Applets

Registration Utility (NewUser.exe)

Some of the Host applets provided in the Developers Package are keyed to allow Innovative to obtain end-user contact information. These utilities allow unrestricted use during a 20 day trial period, after which you are required to register your package with Innovative. After, the trial period operation will be disallowed until the unlock code provided as part of the registration is entered into the applet. After using the NewUser.exe applet to provide Innovative Integration with your registration information, you will receive:



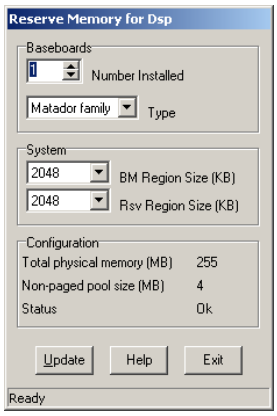
The unlock code necessary for unrestricted use of the Host applets

A WSC (tech-support service code) enabling free software maintenance downloads of development kit software and telephone technical hot line support for a one year period.

Reserve Memory Applet (ReserveMemDsp.exe)

Each Innovative PCI-based DSP baseboard requires 2 to 8 MB of memory to be reserved for its use, depending on the rates of bus-master transfer traffic which each baseboard will generate. Applications operating at transfer rates in excess of 20 MB/sec should reserve additional, contiguous busmaster memory to ensure gap-free data acquisition.

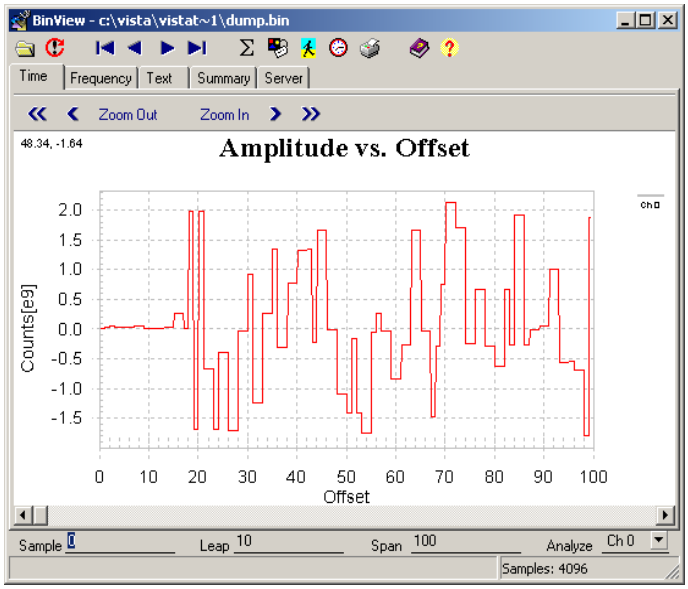
To reserve this memory, the registry must be updated using the ReserveMemDsp applet. If at any time you change the number of or rearrange the baseboards in your system, then you must invoke this applet to reserve the proper space for the busmaster region. See the Help file ReserveMemDsp.hlp, for operational details.



Data Analysis Applets

Binary File Viewer Utility (BinView.exe)

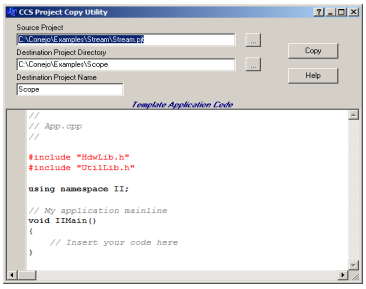
BinView is a data display tool specifically designed to allow simplified viewing of binary data stored in data files or a resident in shared DSP memory. Please see the on-line BinView help file in your Binview installation directory.



Target Programming Applets

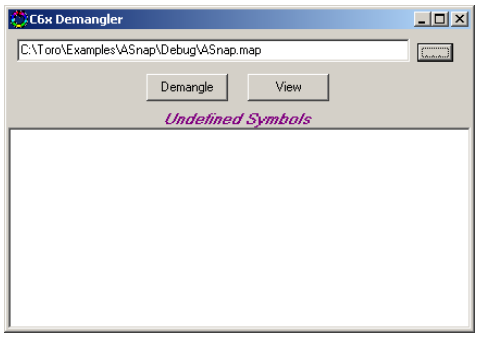
Target Project Copy Utility (CopyCcsProject.exe)

The CopyCcsProject.exe applet is used to copy all project settings from a known-good template project into a new DSP Code Composer project. This simplifies new project development, by eliminating the multi-step process of copying the myriad individual project settings from a source project in a newly-created project.



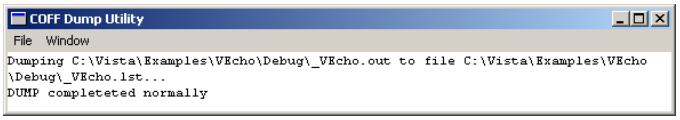
Demangle Utility (Demangle.exe)

The Demangle applet is designed to simplify use of the TI dem6x.exe command-line utility. When building C++ applications, the built-in symbol mangler in the TI compiler renders symbolic names unreadable, such that missing or unresolved symbol errors displayed by the linker no longer correlate to the symbol names within your code. To work around this limitation, enable map file generation within your CCS project. Then, browse to the map file produced by the linker using the Demangle utility. The utility will display proper symbol names for all unresolved externals.

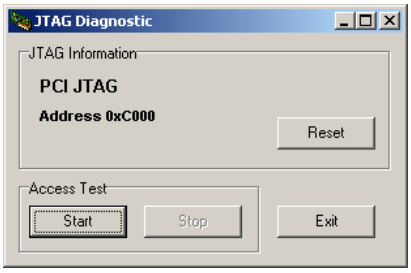


COFF Section Dump Utility (CoffDump.exe)

CoffDump.exe parses through a user-selected COFF file stored on the hard disk and ascertains the complete memory consumption by the DSP program. Memory usage for each of the sections defined in the applications command file are tabularized and the results are written to the Windows NotePad scratch buffer.



JTAG Diagnostic Utility (JtagDiag.exe)



JtagDiag.exe is used to re-initialized the JTAG scan-path interface which connects the Code Hammer debugger's PCI plug-in board with the target DSP. Use this utility prior to invoking Code Composer Studio, to insure that the communications link is viable and clear. This utility is also convenient in confirming that the Code Hammer installation is complete and correct.

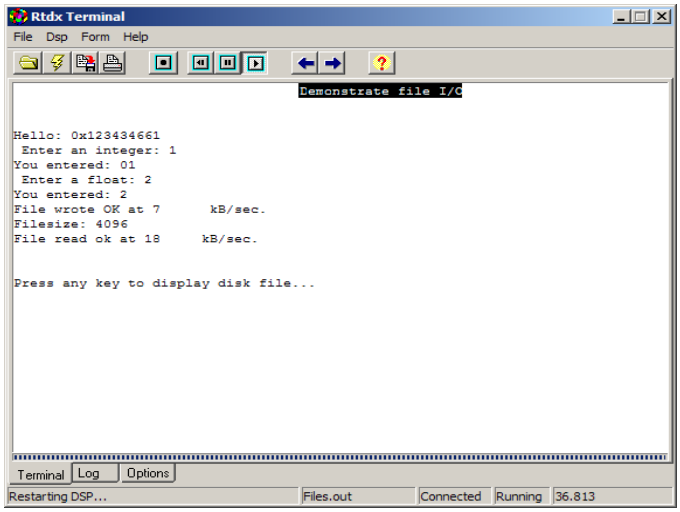
RtdxTerminal - Terminal Emulator

This applet provides a C++ language-compatible, standard I/O terminal emulation facility for interacting with the TermIo library running on an Innovative Integration target DSP processor. Display data is routed between the DSP target and this Host the terminal emulator applet in which ASCII output data is presented to the user via a terminal emulation window and host keyboard input data is transmitted back to the DSP. The terminal emulator works almost identically to console-mode terminals common in DOS and Unix systems, and provides an excellent means of accessing target program data or providing a simple user interface to control target application operation during initial debugging.

RtdxTerminal is implemented as an out-of-process extension to Code Composer Studio. Consequently, it must be used in conjunction with CCS and a JTAG debugger - it cannot operate stand-alone.

The terminal emulator is straightforward to use. The terminal emulator will respond to stdio calls automatically from the target DSP card and should be running before the DSP application is executed in order for the program run to proceed normally. The DSP program execution will be halted automatically at the first stdio library call if the terminal emulator is not executing when the DSP application is run, since standard I/O uses hardware handshaking. The stdio output is automatically printed to the current cursor location (with wraparound and scrolling), and console keyboard input will also be displayed as it is echoed back from the target.

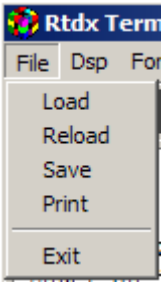
The terminal emulator also supports Windows file I/O using the TermFile library object.



Important Note:
Before using the terminal emulator, you must register your Pismo Toolset. Until you do so, usage will be restricted to a 20-day trial period for the terminal emulator and other applets contained in the Toolset. To register, fill out the contents of the Registration Form, then click on the Register Now button. This will print a Registration report which, must be faxed to Innovative Integration. Innovative Integration will E-mail you an Access Code, which must be typed into the Registration Form for all the features to be enabled.

Terminal Emulator Menu Commands
The terminal emulator provides several menus of commands for controlling and customizing its functionality. These functions are available on the menu bar, located at the top of the the terminal emulator main window. Speed button

equivalents for each of the menu options are also available on the button bar located immediately beneath the menu bar. The following is a description of each menu entry available in the terminal emulator, and its effects.



The File Menu

File | Load - provides for COFF (Common Object File Format) program downloads from within the terminal emulator. When selected, a file requester dialog box is opened and the full pathname to the COFF filename to be downloaded is selected by the user. Clicking "Open" in the file requester once a filename has been selected will cause the requester to close and the file to be downloaded to the target and executed. Clicking "Cancel" will abort the file selection and close the requester with no download taking place.

This operation can optionally be initiated via the button.

File | Reload - Reloads and executes the COFF file last downloaded to the target. It provides a fast means to re-execute the application program most recently loaded into the target board.

This operation can optionally be initiated via the button.

NOTE: File | Load and File | Reload functions use the JTAG debugger and Code Composer Studio in order to effect the program download.

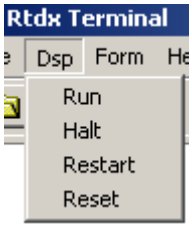
File | Save – saves the textual contents of the Terminal and Log tabs to a user specified file.

File | Print - prints the textual contents of the Terminal and Log tabs to a user specified printer.

File | Exit – closes the emulator application, terminating console emulation.

The DSP Menu

Dsp | Run - causes the terminal emulator to bring the target board into a cold-start, uninitialized condition. This is functionally identical to performing Debug | Run within Code Composer Studio.



This operation can optionally be initiated via the button.

Dsp | Halt - causes the terminal emulator to suspend DSP program execution. This is functionally identical to performing Debug | Halt within Code Composer Studio.

This operation can optionally be initiated via the button.

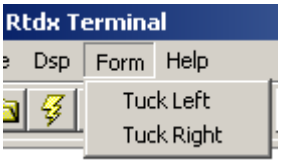
Dsp | Restart - rewinds the DSP program counter to the application entry point, usually *c_int00()*. This is functionally identical to performing Debug | Restart within Code Composer Studio.

This operation can optionally be initiated via the button.

Dsp | Reset - causes the terminal emulator to bring the target board into a cold-start, uninitialized condition. This is functionally identical to performing Debug | Reset Dsp within Code Composer Studio.

This operation can optionally be initiated via the button.

The Form Menu



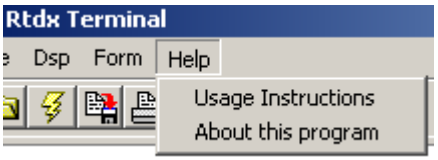
Form | Tuck Left - repositions the main application window to the bottom left of the Windows desktop.

This operation can optionally be initiated via the button.

Form | Tuck Right - repositions the main application window to the bottom right of the Windows desktop.

This operation can optionally be initiated via the button.

The Help Menu



Help | Usage Instructions - displays online help detailing use of the application, including command-line arguments.

This operation can optionally be initiated via the button.

Help | About this Program - displays a dialog containing program revision and tech support contact information.

Options Tab:

The Options tab (seen below) contains controls to allow user-customization of the appearance and operation of the terminal emulator.

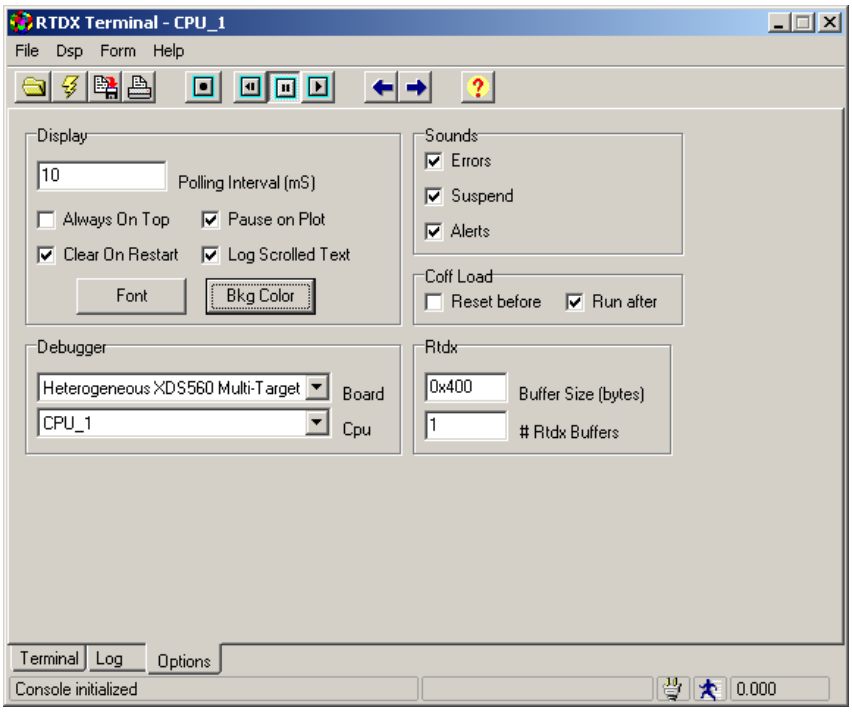


Figure 21. RtdxTerminal Options

Display *Group*
Controls within the Display group box govern the visual appearance of the terminal emulator, as detailed below.

Polling Interval - specifies the period, in milliseconds, between queries for data received from the DSP via the JTAG RTDX interface. Lower numbers increase performance but increase Host CPU load.

Always on Top - specifies that the terminal application should always remain visible, atop other applications on the Windows desktop. This check box controls whether the terminal emulator is forced to remain a foreground application, even when it loses keyboard focus. This is useful when running stdio-based code from within the Code Composer environment, when it's preferable to make terminal visible at all times. The terminal will remain atop other windows when this entry is checked. Select the entry again to uncheck and allow the terminal emulator window to be obscured by other windows.

Clear on Restart - specifies whether the terminal display and log will be automatically cleared whenever the DSP is restarted.

Pause on Plot - specifies whether standard I/O will be suspended following display of graphical information in the Binview applet which is automatically invoked via use of the Pismo library

Plot() command. If enabled, standard I/O may be resumed by clicking the button.

Log Scrolled Text - specifies whether text information which scrolls off screen on the Terminal tab is appended to the Log display. If enabled, standard I/O performance will degrade slightly during lengthy text outputs.

Font - button invokes a font-selection dialog which allows selection of user-specified font within the Terminal and Log text controls.

Bkg Color - button invokes a color-selection dialog which allows selection of user-specified background color within the Terminal and Log text controls.

Sounds Group
Controls within the Sounds group box govern the audible prompts generated by the terminal emulator, as detailed below.

Errors - if enabled, file I/O and other errors encountered during operation generate an audible tone.

Suspend - if enabled, suspension of standard I/O, such as following plotting via Binview, generate an audible tone.

Alerts - if enabled, alert conditions encountered during standard I/O, such as upon display of the ASCII bell character, generate an audible tone.

Coff Load Group
Controls within the Coff Load group box govern behavior surrounding a COFF executable download.

Reset Before - if enabled, the Code Composer Debug | Reset DSP behavior is executed before attempting to download the user-specified COFF file.

Run After - if enabled, the Code Composer Debug | Run behavior is executed immediately following the download of a user-specified COFF file.

Debugger Group
Controls within the Debugger group box specify the target DSP with which RTDX communications is established.

Board - specifies the board hosting the target DSP to be used in RtdxTerminal stdio communications. This combo box is populated with all available board types configured using the Code Composer Setup utility.

Cpu - specifies the identifier of the specific DSP to be used in RtdxTerminal stdio communications. This combo box is populated with all available CPUs present on the baseboard as configured using the Code Composer Setup utility.

Terminal Emulator Command Line Switches
The terminal emulator also provides the following command line switches to further modify program behavior. The switches must be supplied via the command line or within Windows shortcut properties (see the Installation section for more information), and will override the default behavior of the applet.

Multiple instances of the terminal emulator may be invoked simultaneously in order to support installations utilizing multiple target boards. Instances of the terminal emulator, after the first loaded instance must be configured via command line switches in order to properly communicate with their associated target.

-**board** *boardtype* - Use the -board switch to force an instance of the terminal emulator to communicate with a specific type of target board, boardtype. Supported board types are those configured using the Code Composer Setup utility, such as "C64xx Rev 1.1 XDS560 Emulator".

-**cpu** *cputype* - Use the -cpu switch to force an instance of the terminal emulator to communicate with a specific CPU on a target board. Supported CPU types are those configured using the Code Composer Setup utility, such as "CPU_1" or "CPU_A".

-**f** *filespec* - Use the -f switch to force the terminal emulator to load and run the specified COFF file. The "*filespec*" field should be a standard Windows file specification, including both the path and file name as a unit, to allow the user to force the terminal emulator to download the specified file to the target DSP board, as soon as the terminal emulator is loaded. This field is particularly useful in situations where the the terminal emulator is "shelled to" from within an other Host applications to facilitate the automatic execution of target applications employing standard I/O.

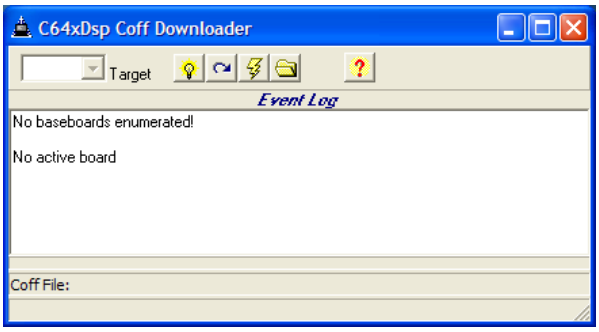
Applets for the C64x DSP Processor

The C64x Processor appears on the PCI Bus as an independent object from the baseboard it resides on. These Applets use this C64x baseboard alone and can be used on any baseboard that possesses a C64x DSP on it.

COFF Downloader (Download.exe)

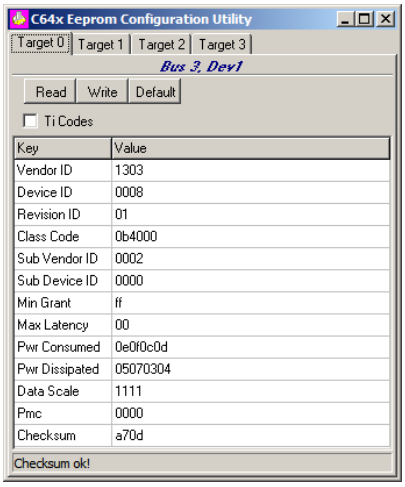
The download applet is used to load known-operational DSP executables to DSP baseboards.

The utility may be used to start DSP applications on PC power-up, through its command-line interface, or to start a DSP application from its GUI Windows user interface. It is also capable of downloading a minimal "boot" application, which is convenient when attempting to start a new Code Composer debug session after having initialized the JTAG scan path with JtagDiag.exe.



ConfigRom: C64x DSP EEProm Configuration Utility (C64xEeprom.exe)

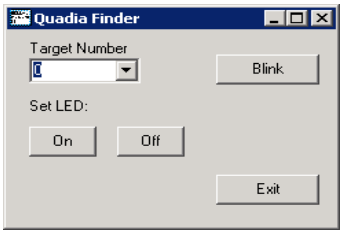
The C64xEeprom applet is used to program the configuration values which are queried by the operating system during enumeration. These values are very rarely changed, and should only be altered under the direction of an Innovative Integration technician.



Applets for the Quadia/Duet Baseboard

Baseboard Finder (Finder.exe)

The board Finder is designed to help correlate Quadia baseboard target numbers against PCI slot numbers in systems employing multiple baseboards.



Target Number
Select the Target number of the baseboard you wish to identify using the Target Number combo box.

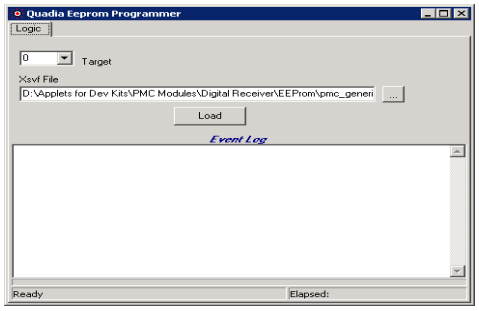
Blink
Click the Blink button to blink the LED on the baseboard for the specified target. It will continue blinking until you click Stop.

On/OFF
Use the On and Off buttons to activate or deactivate (respectively) the LED on the baseboard for the specified target.

When you exit the application, the board's LED will remain in the state programmed by this applet.

PCI Logic Update Utility (Eeprom.exe)

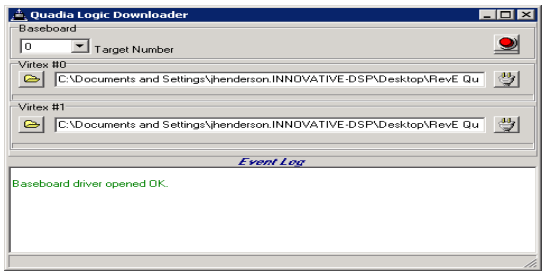
The Logic Update Utility applet is designed to allow field-upgrades of the logic firmware on the Quadia baseboard. The utility permits an embedded firmware logic update file to reprogrammed into the baseboard Flash ROM, which stores the "personality" of the board. Complete functionality is supplied in the application's help file.



Logic Download Utility (LogicLoader.exe)

The logic download applet is used to deliver known-operational logic images to either of the logic devices installed on a Quadia baseboard. The utility may be used to configure firmware either through its command line interface or from its GUI Windows user interface. The former is often convenient during PC boot-up.

This application supports configuration of the on-board Virtex logic device from an EXO file produced by popular logic design tools (including Xilinx's). It is essential that the Virtex be programmed before attempting to download COFF images to the DSP, since some of the baseboard peripherals are dependent on the personality of the configured logic.



Chapter 11. *Target Peripheral Devices*

This chapter describes the peripheral devices on the Quadia and Duet and their use. An overview of these cards has been presented in the Target Basics chapter and this should be reviewed first for general orientation. The peripherals functionality described in this manual is for the FrameWork Logic for Quadia and Duet; customizing the logic to implement new features is described in the *FrameWork Logic User Guide*.

As a starting point, the Quadia and Duet memory maps for both PCI and the DSPs are presented as delivered on the standard hardware. This is followed by a discussion of the hardware that refers to the memory map.

For the sake of general organization, let's divide the peripherals into a few big categories: card controls, communications and IO. Card controls covers the use of interrupts on the card, reset control and synchronization. Communications covers the on-card and off-card communications ports including PCI. Finally, the IO section describes the PMC modules and other IO devices that are part of Quadia or Duet.

Quadia Memory Map

Each Quadia and Duet enumerates five and three devices respectively, on the PCI bus – the baseboard resources are separate from each of the four DSPs (two for Duet). Further, the baseboard maps its control registers and its dual port ram into a separate regions. Similarly, each DSP maps its control registers and external memory into separate regions. So in total, four (two for Duet) separate regions are mapped into PC memory space for each baseboard installed.

Quadia/Duet Control Registers

The PCI memory map below is for the baseboard control registers only, not the DSPs. When Quadia or Duet enumerates in the standard configuration, the system will show the four DSPs (two for Duet) plus the Quadia or Duet baseboard. This memory map is for the Quadia and Duet baseboards. For the DSP memory map, refer to the TI documentation (SPRU190).

PCI Memory WORD Address Base + (hex)	PCI Memory Address Base + (hex)	Logic Address (hex)	Read/ Write	Description
0x00000000	0x00000	0x00	R/W	Control/Status word
0x00000014	0x00014	0x05	W	Cluster0 configuration data register
0x0000005C	0X0005C	0x17	W	Cluster1 configuration data register (Unused for Duet)
0x00000018	0X00018	0X06	R/W	Cluster 0 control register
0x0000001C	0X0001C	0X07	R	Cluster 0 status register

PCI Memory WORD Address Base + (hex)	PCI Memory Address Base + (hex)	Logic Address (hex)	Read/ Write	Description
0x00000020	0X00020	0X08	R/W	Cluster 1 control register (Unused for Duet)
0x00000024	0X00024	0X09	R	Cluster1 status register (Unused for Duet)
0x00000028	0X00028	0X0A	W	PLL 0 Data Register
0x0000002C	0X0002C	0X0B	W	PLL 1 Data Register (Unused for Duet)
0x00000044	0X00044	0X11	R/W	DSP0 CONTROL REGISTER
0x00000048	0X00048	0X12	R/W	DSP1 CONTROL REGISTER
0x0000004C	0X0004C	0X13	R/W	DSP2 CONTROL REGISTER (Unused for Duet)
0x00000050	0X00050	0X14	R/W	DSP3 CONTROL REGISTER (Unused for Duet)
0x000000D0	0X000D0	0X34	W	PLL1 CONTROL REGISTER
0x000000D4	0X000D4	0X35	W	PLL0 CONTROL REGISTER (Unused for Duet)

Table 15. Quadia/Duet Baseboard PCI Memory Map (BAR 1)

The register definitions show the bit fields and their use.

PCI Control Register

This register gives the control bits for the baseboard. Bits 1..4 are associated with the global memory pool controls.

Bit	Function	Default State after Reset
0	LED indicator on front panel	0 = off
1	Velocia FPGA DLL reset for global memory pool clock	1= reset
2	Start Global Memory Pool initialization	0=no init
3	Global Memory Pool Clock Enable	0=no clock
4	Global Memory Pool refresh enable	0 = no refresh
5	Cluster 0 FPGA reset , 1= reset 0 = no reset	0= no reset
6	Cluster 1 FPGA reset , 1= reset 0 = no reset	0= no reset
7	Global Trigger 1= on 0= off	0 = off
8..31	Not used	-

Table 16. Quadia/Duet Baseboard Control Register (Write, BAR1 +0x0)

PCI Status Register
This register gives status for baseboard devices. The INIT and DONE pins of the cluster FPGAs are read back during the FPGA loading process. The PLL busy and enable are used to monitor the serial interface from the Velocia FPGA to the PLL devices. The DCM lock bits are used in the Global Memory Pool controller and indicate when the clocks are stable and ready for use.

Bit	Function	Value
0	PLL busy indicator. If true (‘1’), then the PLL shift register is busy sending data to a PLL. Do not write data to the PLL registers until this is false.	0 = not busy
1	PLL Load enable. This bit indicates which PLL receives the data sent over the PLL serial interface.	0 = PLL 0
2	Cluster 0 FPGA Done. When true (‘0’), this bit indicates a successful load to the FPGA. This is the DONE pin on the Xilinx VP40 for cluster 0.	‘1’ = FPGA is NOT loaded
3	Cluster 0 FPGA Init. This bit is monitored during the loading process. If it is ever true (‘1’) during the load, then a checksum error has occurred.	0 = no checksum error
4	Cluster 1 FPGA Done. When true (‘0’), this bit indicates a successful load to the FPGA. This is the DONE pin on the Xilinx VP40 for cluster 1.	‘1’ = FPGA is NOT loaded
5	Cluster 0 FPGA Init. This bit is monitored during the loading process. If it is ever true (‘1’) during the load, then a checksum error has occurred.	0 = no checksum error
6	Global memory pool controller external DCM locked	0 = not locked
7	Global memory pool controller internal DCM locked	0 = not locked
8...31	Not used	-

Table 17. Quadia/Duet Baseboard Status Register (Read, BAR1 + 0x0)

Cluster 0 FPGA Control Register
This register is used to load the Cluster 0 FPGA image into the device over its SelectMap interface.

Bit	Function	Default State after Reset
0	Enable Cluster 0 FPGA image loading.	0 = not enabled
1	Cluster 0 FPGA reset.	1 = reset
2	Cluster 0 FPGA configuration memory clear. This bit drives the INIT pin of the FPGA and is used during the FPGA loading process.	‘0’ = memory clear off
3...31	Not used	-

Table 18. Cluster 0 FPGA Control Register (write, BAR1 + 0x18)

Cluster 1 FPGA Control Register
This register is used to load the Cluster 1 FPGA image into the device over its SelectMap interface. Note that Duet only has one Cluster, thus following table table is not applicable for Duet.

Bit	Function	Default State after Reset
0	Enable Cluster 1 FPGA image loading.	0 = not enabled
1	Cluster 1 FPGA reset.	1 = reset
2	Cluster 1 FPGA configuration memory clear. This bit drives the INIT pin of the FPGA and is used during the FPGA loading process.	'0' = memory clear off
3...31	Not used	-

Table 19. Cluster 1 FPGA Control Register (write, BAR1 + 0x20)

DSP Control Registers
These registers are used to reset the DSPs. There is one register for each DSP. There are four DSPs for Quadia and two for Duet.

Bit	Function	Default State after Reset
0	Reset the DSP	1 = reset
1...31	Not used	-

Table 20. DSP Control Registers (write, DSP0 = +0x44, DSP1 = +0x48, DSP2 = +0x4C, DSP3 = +0x50)

Cluster FPGA Status Registers
These registers are used to monitor the cluster FPGA configuration process. There is one register for each FPGA. Duet has one whereas Quadia has two FPGA Clusters.

Bit	Function	Value
0	FPGA DONE indicates that the FPGA configuration process was successful.	1 = configuration successful
1	FPGA INIT indicates whether a FPGA initialization and configuration is good.	1 = OK, 0 = bad
2...31	Not used	

Table 21. Cluster FPGA Status Registers (read, Cluster0 = +0x1C, Cluster1 = +0x24)

DSP Memory Map

The DSP has two external buses mapped into its memory space: EMIF A and EMIF B. This memory map shows the decoding for those two external buses that are specific and have been implemented on the baseboards. The memory map for the DSP internal devices can be found in the TI documentation (SPRU190).

Address	CE Space	Read/Write	Function
0x80000000	ACE0	R/W	SDRAM
0x90000000	ACE1		-
0xA0000000	ACE2		-
0xB0000000	ACE3		-

Table 22. DSP EMIF A Memory Map

The only device mapped to EMIF on the baseboard design is SDRAM on CE0. This is 64-bit wide memory.

DSP EMIF B is a 16-bit wide bus that is the primary connection to the cluster FPGA. The memory map for the example Framework logic is shown with register definitions in the *FrameWork Logic User Guide.*

Card Controls

Reset Control

There are multiple resets on Quadia and Duet that either originate with the host system or are software controlled. This is intended to give the application enough control over the baseboard devices to initialize them at the proper time, guard against unexpected behavior and allow the software to return the devices to a known state when needed.

Host Reset
A reset over the PCI bus from the host system is the highest priority reset and will return the Quadia or Duet to a reset state for all DSPs, FPGAs and other devices on the card. Since the PCI reset may indicate a power failure condition or other serious event, the baseboard should be reset as well. In most systems, the host computer would also enumerate the system devices again and this affects any software interacting with the host.

PCI bridges are also reset so any data in their internal queue is lost when a host reset occurs.

The host reset is not generally under program control.

DSP Resets
The DSPs have individual reset controls that are under software from a PCI mapped register. See the memory map section of this chapter for the address and bit definitions. The DSP reset is connected directly to the DSP reset pin. When the DSP is in reset, the PCI bus is active so that programs can be loaded into internal memory. The DSP reset is asserted whenever the PCI bus reset is asserted. If the PCI bus reset occurs, the DSPs will remain in reset until they are released by a software access to the DSP Control Register.

Applications may assert DSP reset to return the DSP to its default state. The Quadia and Duet host software provides methods for controlling resets as part of the application.

PLL Resets
Both of the Quadia baseboard PLLs (one PLL for Duet) can be reset via software control through the PCI mapped register. When the PLL reset is asserted, the PLLs will stop and must be reloaded with their control values before resuming. The PLL reset is asserted whenever the PCI bus reset is asserted and must be de-asserted by software after re-initialization to resume operation.

The Quadia/Duet development software provides methods for PLL control and reset.

FPGA Resets
The cluster FPGAs can be reset via software via the FPGA control registers through a PCI mapped register. This effect of this reset is dependent on the FPGA design. For the standard Framework logic, the FPGA reset is the primary reset to the device and results in all functions being reset. Note that since this is a PCI-mapped register, any PCI device can assert this reset - including the DSPs on the baseboard.

The Quadia/Duet development software provides methods for FPGA control and reset.

DSP Interrupts

Each ‘C6416 processor implements five interrupt input pins which allow external hardware events to directly trigger software or DMA activity. In addition to the interrupt pins, four GP (General Purpose) pins on the process may be used as DMA interrupts. Collectively, these are the interrupts supported to each DSP for program DMA interrupts.

The interrupt assignments in the Framework example logic are shown in the following table.

INT	Interrupt Source	INT Type
4	DSP0 & 2: Software Select DSP1 & 3: Not used	INT/DMA
5	DSP0 & 2: Software Select DSP1 & 3: Not used	INT/DMA
6	FIFOLink Write FIFO 0	INT/DMA
7	FIFOLink Read FIFO 0	INT/DMA

INT	Interrupt Source	INT Type
GP0	FIFOLink Write FIFO 1	DMA
GP1	FIFOLink Read FIFO 1	DMA
GP2	FIFOLink Write FIFO 2	DMA
GP3	FIFOLink Read FIFO 2	DMA
NMI	DCM Lock	INT

Table 23. DSP Interrupt Assignments

Custom logic designs can use the eight interrupts (NMI is not usually used) for any purpose. The GPx pins are general purpose IO on the DSP and must be configured for use as DMA interrupts. Since the GPx pins cannot be used as a software interrupt, these are commonly assigned to FIFOs for communications data flow. Custom designs may also need to condition interrupts for polarity and timing to meet the DSP requirements as described in the TMS320C6416T data sheet.

DSP INT4 and DSP INT5 Source Assignment

DSP interrupts INT4 and INT5 inputs are supported in the Framework logic with a set of control registers and multiplexers that allow application software to dynamically select the source of the signal which will drive each particular interrupt input. The selection codes are identical for INT4 and INT5.

Source[15..0]	Interrupt Source
0	PMC0 INT A
1	PMC0 INT B
2	PMC0 INT C
3	PMC0 INT D
4	StarFabric INT A (Quadia Rev C only)
5	StarFabric INT B (Quadia Rev C only)
6	StarFabric INT C (Quadia Rev C only)
7	StarFabric INT D (Quadia Rev C only)
8	PMC1 INT A
9	PMC1INT B
A	PMC1 INT C
B	PMC1 INT D

Table 24. DSP INT4/5 Source Selection Register

Card Synchronization Features

Quadia and Duet has a number of features for system synchronization requirements including clock generation and synchronization IO. This allows designers to integrate Quadia or Duet in with other signal acquisition hardware and system elements.

Clock Generation
Quadia has two PLLs (one for Duet) on the card for precision clock generation, plus the DSP timers. The most common requirements are to clock the Rocket IO link communications and to provide sample rate clocks to the cluster FPGAs (one FPGA for Duet). The PLLs may be used for either requirement. Use of the PLLs is described in this chapter under PLL use.

Clocks can be sourced from Quadia for use by other cards in the system via the external IO connectors J1 or J2 for cluster 0 and 1 respectively. In custom logic designs, these signals may be directly driven from the FPGA so that clocks in the FPGA may be used elsewhere. It is also possible in custom logic designs to have Quadia supply the clocks to PMC modules over the J4 or spare connections to the PMC.

Synchronizing to external clocks
Quadia has an input for each cluster on the front panel that can be used as an external clock input or as a reference clock to the PLL The input is through an SMB connector J1 or J1 for cluster 0 and 1 respectively. This input is directly connected to the FPGA so that custom designs can directly use this clock in logic designs. Input is expected to be LVTTL (3.3V max) into the FPGA.

It is possible to synchronize to an external clock in the 10 to 25 MHz range by inputting the external input into the PLLs as a reference clock. The PLL can then be used to generate higher frequency that will be synchronized to that clock. The PLL will go a long way to cleaning up the clock for noise, but be aware that the input clock should be good enough to make a good clean clock that is required by Rocket IO or precision analog. A reasonable expectation is that the PLL will clean up the clock by 20 dB at best.

Clock and Trigger Sharing
Clocks and triggers may be shared to other cards using either the J1 and J2 SMBs on the front panel, or by using the signals provided on the CJ4. CJ4 has 12 signal pairs (12 total wires), plus 32 signals directly from each cluster FPGA that may be used in custom logic designs as IO for sharing triggers to other cards. These connections are available on The rear terminal of Quadia on compact PCI connector J5. The connector type and pinout for CJ4 is provided in the appendix of this manual.

The front panel IO connectors J1 and J2 may also be used to connect to other cards in custom logic designs. The signaling standard is LVTTL for these two signals.

Sharing clocks, triggers and other system signals to the PMC sites is accomplished using the J4 connections or spare connections to the PMC in custom logic designs. These can be LVTTL or LVDS signals. It is recommended that LVDS be used for best signal quality and noise rejection. The Framework logic does not implement this sharing.

Communications

One of the primary features of Quadia and Duet that supports real-time signal processing is the tight communications between the DSPs, FPGAs and IO devices. Two separate communications “planes” are provided: PCI bus connectivity and

Data Plane Connectivity. The communications planes are separate, independent entities that allow data to flow freely to each device on the baseboard.

The PCI and Data Plane serve separate purposes in many cases- that of system control and management for PCI and that of real-time data streams for the Data Plane. This is done because the real-time demands of signal processing for low latency, deterministic, high rate data are often difficult to achieve with PCI when it is connected to the system PCI. The system PCI is usually encumbered with random data traffic that in many cases makes it impossible to real-time requirements. The Data Plane has no such encumbrance since it is entirely dedicated to the real-time data and is under design control of the system engineer.

System Level Data Communications Design

System level design with Quadia and Duet begins with an examination of the required data flows and the communications required between devices. Real-time data is the most difficult to manage in most cases, so it is best to begin by identifying the real-time data flows and using the Data Plane for those. The command and control communications between DSPs or with the host system is best handled over PCI since it is flexible and connected to all the DSPs.

The data rates may also quickly define what is possible for connectivity. The following table gives the maximum data rates over the paths on Quadia.

Path	Max Rate (MB/s)	Practical Rate (MB/s)	Restrictions
DSP to DSP over PCI	132	80 with good data buffering	PCI traffic may limit instantaneous rate and availability Arbitration across multiple bridges adds to indeterminacy
DSP to DSP in the same cluster	266	200	Depends on source and destination DSP DMA arbitration may cause indeterminacy More deterministic than PCI
FPGA to FPGA over RIO	200 per link 4 links in each direction	190	Point-to-point connectivity 8b/10b encoding usually required Deterministic Fixed latency possible
FPGA to DSP	266	180 to internal RAM	Depends on destination DSP DMA arbitration may cause indeterminacy
FPGA to SFP	200 per link	190	Point-to-point connectivity 8b/10b encoding usually required Deterministic Fixed latency possible Fiber required for highest rates
PMC to DSP over PCI	132	80	PCI traffic may limit instantaneous rate and availability Arbitration across multiple bridges adds to indeterminacy

Path	Max Rate (MB/s)	Practical Rate (MB/s)	Restrictions
PMC to/from Host	264	160	PCI traffic may limit instantaneous rate and availability Arbitration across multiple bridges adds to indeterminacy
DSP to Host	132	80	PCI traffic may limit instantaneous rate and availability Arbitration across multiple bridges adds to indeterminacy
PMC to FPGA over J4	350	350 Assume 83 MHz clock, faster may be possible if LVDS is used	Dedicated link give low latency, deterministic data Custom protocol in most cases Custom FPGA design required
Host to/from Global Memory Pool	564	400	PCI traffic may limit instantaneous rate and availability Arbitration across multiple bridges adds to indeterminacy Rate is limited by host PCI bus clock rate
StarFabric to/from DSP (Quadia Rev C only)	132	120	Private StarFabric link can be deterministic Latency can be deterministic only in dedicated link mode
XMC Links to Cluster FPGAs	4 links at 200 MB/s each= 800 MB/s	720 MB/s	Private links can be deterministic Highest rate transfers to PMC/XMC modules

Table 25. Maximum Data Rates

As is shown in the table, the highest data rates with the lowest latency and best determinacy are supported between FPGAs and over the SFP connections. These point-to-point connections are best used for the high rate real-time data in most applications.

PCI Bus

The PCI bus is presented in detail in the Target Basics chapter. Refer to that discussion to understand the PCI architecture and features. In this section, the PCI bus is discussed from an application viewpoint involve system architecture.

The PCI bus architecture of Quadia and Duet gives the card a backbone for card control, coordination and data communications that complements the Data Plane. One of the great things about the baseboard PCI bus architecture is that the DSPs, PMCs and Velocia FPGA all have access from the host computer and to one another. This gives great flexibility in coordinating and controlling the data processing that the DSPs are performing without interfering with that process. Hence the fundamental division of command and control on the PCI bus from data processing on the Data Plane.

Using PCI for Command and Control

Each DSP, PMC, Memory Pool and Velocia FPGA enumerates on the PCI bus when the standard bridge configuration is used. This allows the host, or any device on the PCI bus, to communicate with the baseboard DSPs, PMC, Memory Pool and Velocia FPGA.

The main functions implemented on the PCI bus are DSP program loading, DSP control and DSP data communications to the host. At boot time, each DSP is loaded over the PCI bus directly over its PCI interface. Software is provided in Pismo for this. Software is also provided that implements a messaging system with the host for control and coordination. Finally, the DSP has a nice PCI interface that can use DMA to transfer data efficiently to the host.

Most PMCs also use the PCI bus for configuration and control. Any PCI device, including the DSP, may therefore be responsible for the PMC (or vice versa). The fact that configuration and control functions tend to be lower rate and less data than the data channel. This makes it natural in many cases that PMC data flows over J4 to the baseboard while the PCI bus is used for configuration and control. The PCI bus bandwidth is preserved and flexible access to the module is provided over PCI.

Using PCI for Data Communications

Since the PCI bus is used by many devices, high rate real-time data is difficult to manage over PCI. The PCI bus may be used by many devices and may not be available to support the data rates required. This can be alleviated with good buffering at the source, but that does not solve the determinacy and latency problems.

One way to improve PCI performance on baseboard for real-time data applications is to configure the bridge chips so that local PCI cluster is not on the main PCI bus. The Intel 31154 bridge chip used on Quadia may be configured to be made opaque under software control. (This advanced feature is provided on the hardware, contact technical support for information on its use.) when the bridge is opaque, the local PCI bus only has local traffic. This removes the problems of traffic interfering with local data traffic within the cluster, thus allowing the DSPs and PMC have the PCI bus to themselves.

PCI Interrupt Mapping

The PCI interrupt mapping on the baseboard is shown in the following table. Per the PCI, interrupts are shared by device enumeration order. All interrupts over PCI are shared not only with devices on the baseboard, but with other system peripherals. On many low end systems the PCI interrupts, INTA-D, are simply wired together since they are open-drain active low on the bus thus effectively having only one physical interrupt on the system bus.

Interrupt	Devices
INTA	PMC0_INTD, PMC_INTD
INTB	PMC0_INTA, PMC_INTA
INTC	PMC0_INTB, PMC_INTB, DSP1 PCI INT, DSP3 PCI INT
INTD	PMC0_INTC, PMC_INTC, DSP0 PCI INT, DSP0 PCI INT

Table 26. PCI Interrupt Assignments

PMC Interrupt Control

The interrupt control logic on the baseboard allow for the PMC interrupts to serviced by the DSPs without involving the host. This allows the DSP to responsible for servicing the PMC and can greatly improve real-time performance since the host operating system is frequently not capable of good response time to interrupts. Under Windows, it is not uncommon for interrupts to take over 1 ms for system response time. The DSP can respond in less than 1 us in most cases.

See the interrupt section for register definitions on DSP interrupts.

StarFabric (Quadia Rev C Only)

StarFabric can be used to integrate Quadia into high performance systems either as a replacement for the PCI bus or as a private data channel. The StarFabric bridge connects to the secondary PCI bus with a 64-bit, 66 MHz capable connection. In the PCI legacy mode, StarFabric simply replaces the PCI bus. Data latency is typically higher than standard PCI since the data is serialized across the data link.

More interesting to system designers however, is the capability to use StarFabric in more advanced modes that allow concurrent data streams across the StarFabric. In this mode, it is possible to achieve good real-time performance since the data links are private and dedicated. Typically a StarFabric routing hub is used to control the fabric switching and set up the data paths. Advanced software is typically needed for this configuration so a discussion with an applications engineer is usually needed. Contact technical support for more information.

Quadia is compatible with PCIMG 2.17 standard for StarFabric systems. The StarFabric connections are over cPCI connector CJ3. See the appendix for pinout details.

Data Plane

The Data Plane, as implemented in The FrameWork Logic on Quadia and Duet, connects the DSPs, FPGAs and IO devices to provide high rate, low latency, deterministic data communications for real-time signal processing. The Data Plane is centered around connections to the FPGA: Rocket IO links between FPGAs, SFP ports, DSPs EMIF B interface, XMC connections and PMC J4 connections. Within the FPGA design, components for each device allow the designer to easily connect the data plane to suit any specific application.

In this section, the implementation in the FrameWork Logic is discussed. The FrameWork Logic provides a mesh of FIFOs for DSP intercommunication and SFP port connections to external devices. The *FrameWork Logic User Guide* discusses this structure in detail and the supporting logic components.

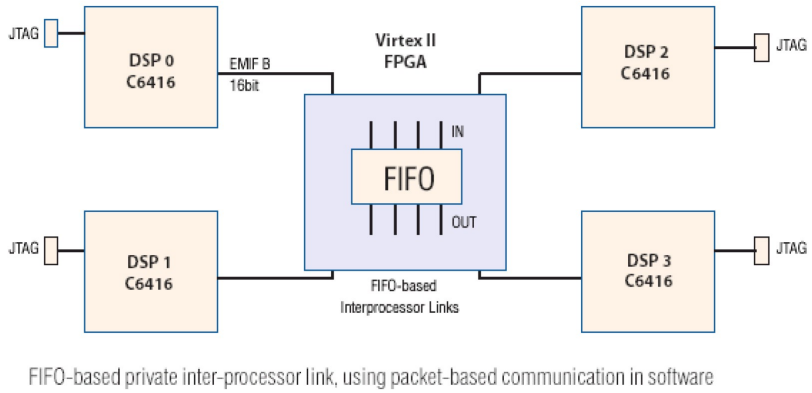


Figure 22. Data Plane Connections on Quadia

DSP FIFOLinks

The FrameWork Logic connects the DSPs with a mesh of FIFOs that connects every DSP to every other DSP over a private link. This provides an unimpeded data path for the DSPs to communicate.

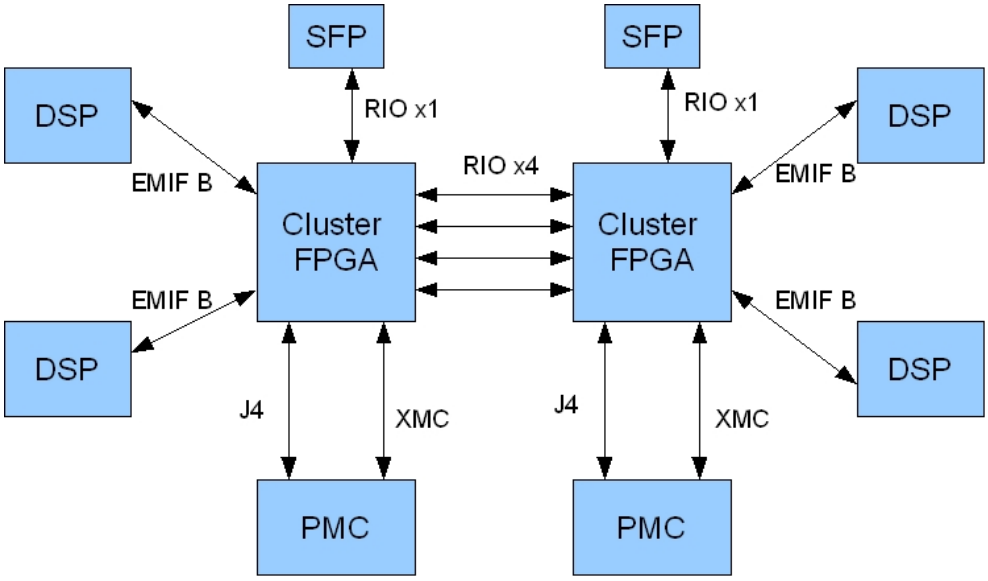


Figure 23. Quadia DSP FIFOLink Mesh

The connections are implemented in the cluster FPGA using FIFOs connected to each DSP EMIF B. Each DSP has three output FIFOs and three input FIFOs connecting it to the other DSPs. Data transfers are paced by programmable threshold levels indicating the data level in the FIFO that are used as either CPU or DMA interrupts. Interrupt assignments are shown in the DSP interrupt discussion section.

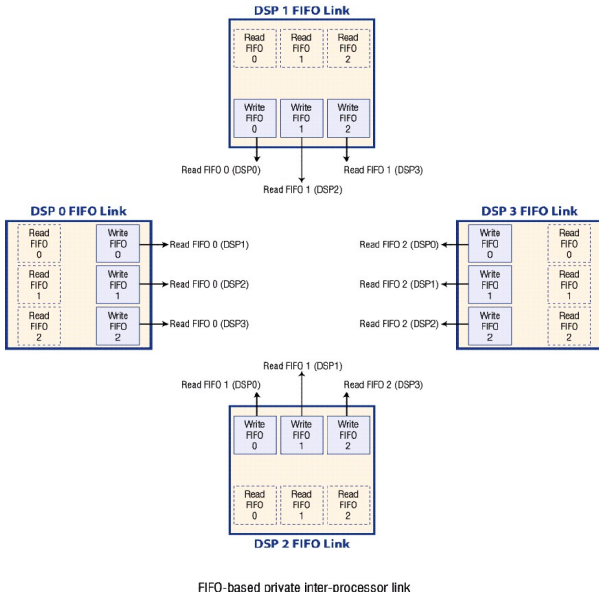


Figure 24. Quadia Data Plane FIFO Mesh

As is shown in the FIFO Mesh diagram, the interconnections are numbered 0,1 and 2 for each DSP but the actual DSP connected on that link depends on which DSP you are programming. For example, if you are programming DSP 1, then your Link 0 (FIFO0) is connected to DSP 0. Here's a table that is a quick reference. Note that only DSP 0 and DSP 1 is available on Duet.

For DSP	Link 0 Connects to	Link1 Connects to	Link2 Connects to
0	DSP 1	DSP 2	DSP 3
1	DSP 0	DSP 2	DSP 3
2	DSP 0	DSP 1	DSP 3
3	DSP 0	DSP 1	DSP 2

Table 27. DSP FIFO Mesh Connections

The Framework logic uses two components in the cluster FPGAs to implement the FIFOLinks in the mesh: a datamover component for two DSPs in the same cluster and an ii_fifo_rio component between FPGAs.

The Framework logic implementation of the DSP to DSP interface component is shown in the following diagram. Each FIFO in the mesh is 1KB and is implemented using Block RAMs in the FPGA. The DSP bus connection is a synchronous burst interface that allows data transfers at up to 266 MB per second. Status and control registers are provided for each DSP. Interrupts are used to pace the data flow using CPU or DMA.

DSP to DSP Data Plane Component

For cluster to cluster communications in Quadia, the Framework logic uses Rocket IO (RIO) links between the FPGAs to between the DSPs. This RIO link component connects two RIO ports together with a FIFO back end. One component is needed in each FPGA to complete the link. Similar to the DSP component, the RIO link component has 1K FIFOs with flow control, interrupts and status monitoring. The flow control in this case between RIO is transmitted over the RIO link automatically to prevent overruns. In custom designs this same component may be used to communicate between FPGAs or to external devices over the SFP ports.

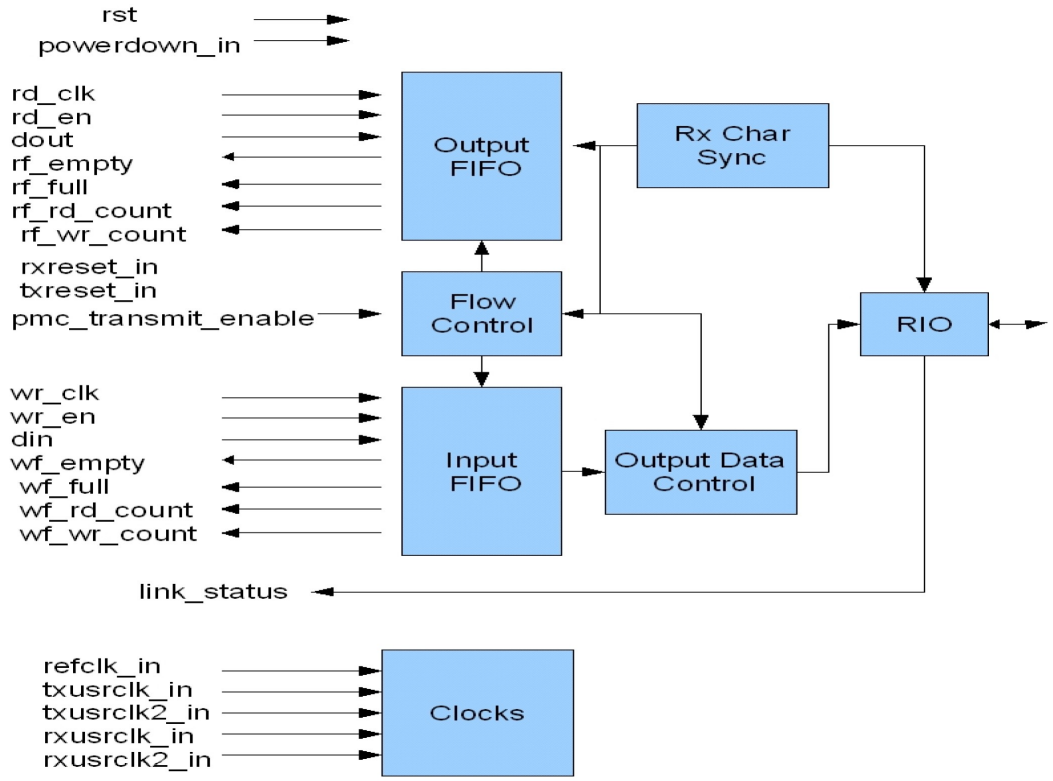


Figure 25. Rocket IO Link Component

DSP FIFOLink FIFO Data

The FIFOLink data is read as a 32-bit value from the FIFOs. These FIFOs are mapped to EMIF B CE0 as shown in the memory map. Normally a DMA controller is used to move the data for efficiency.

DSP FIFOLink FIFO Status Registers

All of the FIFOLinks have a status register for each FIFO for flow control and monitoring by the DSP. In the example logic, there are three FIFOLinks for each DSP with a bidirectional FIFO with a read and write status register.

Bit	Function	Value
7..0	FIFO Count. The number of 16-bit words in the FIFO.	0 to 255
8	FIFO Full	1 = full
9	FIFO Empty	1 = empty
10	FIFO Threshold exceeded. When the FIFO level exceeds the programmed level, this bit is true. This signal can signal an interrupt if so configured in the interrupt control register.	1 = threshold exceeded
31...11	Not used	-

Table 28. FIFOLink Status Registers

DSP FIFOLink FIFO Control Registers

All of the FIFOLinks have a control register for each FIFO to set the interrupt threshold and the "burst count" for interrupts to the DSP. The threshold gives the level at which the DSP interrupt will be signaled. The burst count is the number of points that will be read before another interrupt is expected. The burst count prevents false interrupts to the DSP as the FIFO moves through the threshold.

Bit	Function	Value
9..0	FIFO threshold and burst count. The number of 16-bit words in the FIFO before an interrupt is signaled for reads, the amount of available space in the FIFO for writes. This is also the number of points that must be read before another interrupt is signaled.	0 to 1023
31...10	Not used	

Table 29. DSP FIFOLink FIFO Control Registers

DSP FIFOLink Reset Control Register

This register gives reset control for the FIFOs in the example Framework logic. This allows the application software to purge the FIFOs

Bit	Function	Default Value
0	DSP FIFO links reset. This resets the FIFOs between DSPs in the same cluster.	1 (reset)
1	Rocket IO Link 0 reset. This resets the logic associated with Rocket IO link 0 between FPGAs.	1 (reset)
2	Rocket IO Link 0 receive reset. This resets the FIFO associated with the Rocket IO link 0 receive.	1 (reset)
3	Rocket IO Link 0 transmit reset. This resets the FIFO associated with the Rocket IO link 0 transmit.	1 (reset)
4	Rocket IO Link 1 reset. This resets the logic associated with Rocket IO link 0 between FPGAs.	1 (reset)
5	Rocket IO Link 1 receive reset. This resets the FIFO associated with the Rocket IO link 0 receive.	1 (reset)
6	Rocket IO Link 1 transmit reset. This resets the FIFO associated with the Rocket IO link 0 transmit.	1 (reset)
15..7	Not used	-

Table 30. DSP FIFOLink Reset Control Register

A software driver for use with the FIFO Mesh is provided in the Pismo Toolset for Quadia. This driver is a DSP/BIOS compliant driver that allows the DSP to efficiently use the FIFO mesh in DSP applications.

SFP Links

Quadia has two SFP ports, one from each cluster DSP. These links are a Rocket IO port from each cluster FPGA with an industry-standard SFP communications physical interface. Links to other cards and external IO devices at up to 2.0 Gbps are supported over these ports. Higher data rates are available with higher speed grade FPGA devices, up to 3.125 Gbps. The data rate also depends on the SFP module used, transmission distance and error rate required. Low cost fiber optic cables may be used for short distances under 10 meters.

Any protocol may be implemented over the SFP links. The SFP does not impose any protocol, only that the data be balanced and continuous so that the clock recovery and line balance are able to function. The Framework logic implements a simple point-to-point link that has a FIFO at each end. This easy to use component provides a simple link to other Quadia cards or can be embellished or adapted to many applications when custom logic work is done. Raw data can be continuously transmitted and received over the links at the full bandwidth – full bandwidth being 200 MB/s per link when 8b/10b encoding is used.

What are SFP Modules
SFP modules are "Small Form-factor Pluggable" interconnects that allow the Rocket IO to be transmitted over copper or fiber optic cables. The modules are protocol agnostic and only give the physical interface. SFP modules are widely used for communications applications such as routers, SANs and hubs. They are available from a number of vendors including 3Com, I2O and MRV Communications.

The SFP modules for Quadia must be electrically compatible with the Xilinx Rocket IO, which use CML (current mode logic) signaling. Not all SFPs use CML, only ones with the SX designation.

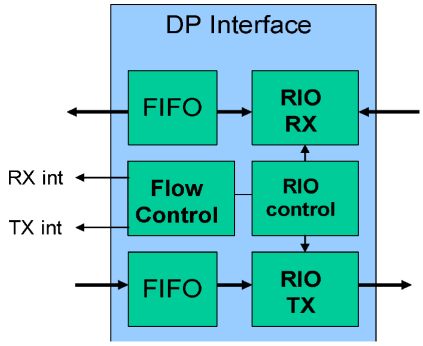
Figure 26. Some SFP modules (Picture courtesy of MRV Communications)

Here are few SFP modules that are compatible with Quadia.

Vendor	Model Number	Interface Type	Data Rate	Mating Cable
3 Com	3CSFP91	Fiber – 1000 Base -SX	1Gbps	Fiber LC-LC, Amp 1374657-x
MRV	SFP-MR27D-SR1	Fiber (1310 nm)	2.7 Gbps	Fiber LC-LC, Amp 1374657-x

Table 31. A Sample of Compatible SFP Modules

SFP Link Component
The Framework logic implements a simple point-to-point connection component that has FIFOs at the back end connected to the DSP EMIF B. The FIFOs are 1K and registers are provided for FIFO control and DSP interrupt control. SFP module status is also reported in a register to monitor.



SFP Data Port
The Framework logic maps the SFP FIFO on EMIF B CE0 at 0x60040000 for both reads and writes. The FIFO interface supports burst accesses in this memory location so that transfer rates of up to 266 MB per second are supported. No data format is implied, although the software treats this as a 32-bit memory device.

SFP Status Register
This register reports the SFP status. These are directly from the SFP module. Specific criteria or cause for each bit to be signaled may vary according to the SFP vendor.

Bit	Function	Value
0	Transmit Fault. The SFP module indicates a transmit failure when this bit is '1'.	1 = fault
1	Loss of Signal (LOS). The SFP module indicates that the receive signal has been lost when this bit is '1'.	1 = receive signal lost
31..2	Not used	-

Table 32. SFP Status Registers

SFP FIFO Status Register
This register shows the FIFO status for the bi-directional FIFO associated with the example SFP link interface logic.

Bit	Function	Value
7..0	Write FIFO count. This is the number of 16-bit words in the write FIFO that will be transmitted from Quadia.	0 to 255
15..8	Read FIFO count. This is the number of 16-bit words in the read FIFO that have been received into Quadia.	0 to 255

Table 33. SFP FIFO Status Register

SFP Cabling
The SFP uses fiber optic cabling. The typical fiber connector is LC type, but various types of fiber are available for transmission distances. For short distances, a 10 meter single mode LC-LC cable can be purchased for about $35. These cables are available in many on-line stores such as Digikey and Fiber Optic Cable Shop.

SFP Error Rates
Some bit errors may occur in data transmissions as bit rate and distance increases. For short distances, these errors are either non-existent or low enough to be ignored. Applications that require error-free communications may need to use higher quality SFP modules and implement error correction as part of the protocol. Be sure to check the error rate of the SFP you select and plan accordingly.

DSP and FPGA Communications

The DSP and FPGA communicate over DSP EMIF B. This is a 16-bit, 133 MHz bus capable of maximum burst rates of 266 MB/s. The Framework logic in the cluster FPGAs decodes the EMIF B memory space for registers and data FIFOs.

Using DSP EMIF B
The DSP EMIF is divided into four parts, referred to as CE spaces. The four CE spaces divisions are signaled on the bus with for CE signals and are supported by the DSP EMIF controller for different memory types and timing. The DSP control signals change with the memory type that must interface with the logic.

It is therefore typical to divide the decoding on the DSP bus into high rate regions and low rate regions. High rate regions such as the main data stream are usually handled, as in the Framework logic, through FIFOs that may be read or written in a continuous data burst. Low rate devices, such as control and status registers, or devices with variable cycle timing are assigned a memory region that is slower and has more relaxed timing for ease of design.

If you examine the memory map for Quadia, you will find that the peripheral devices defined in the logic follow the high speed/ low speed organization. FIFOs have been assigned to CE0 and CE1, whilst registers are defined on CE2 and CE3. CE0 and CE1 are defined as burst memory regions by the EMIF control registers (see Target Basics chapter for values) and CE2 and CE3 are defined as asynchronous memory regions. The burst memory interface has only a latency parameter that is at the discretion of the designer; no other timings are adjustable in this region. The asynchronous memory interface is much more flexible and allows for setup, active and hold timings to be adjusted, but is much slower because only a single data point is transacted each cycle.

The Texas Instruments Peripheral Guide for the C6000 family (SPRU190) gives a complete register definition and explanation of the EMIF controller. Exact cycle timings for the 1 GHz DSP are in the TMS320C6416T data sheet from TI.

Controlling Data Flow to the DSPs

The DSPs are most efficiently used when data is moved by DMA. Primary, high rate data paths should be implemented using DMA controllers to move the data to/from the FPGA. This leaves the CPU available to continue processing. The '6416 architecture is has an efficient memory cache controller that allows the CPU and DMA to run concurrently in many cases. Using DMA to move the data saves valuable computing cycles for signal processing which is why you bought this card. Lower rate setup, control and status is less critical and is typically done by the CPU for convenience.

The Framework logic uses interrupts and status registers to control the data flow. Most high rate data flows are paced by data availability in a FIFO - when enough data is in the FIFO (or there is space in the FIFO for the other direction) then an interrupt is generated to drive a DMA channel on the DSP. It is necessary to only signal an interrupt when the FIFO threshold is exceeded then wait until a fixed number of points is moved out of the FIFO. False interrupts may be triggered if the interrupt is allowed to simply track the FIFO level since it may transition through the threshold level many times during the data moving operation since both sides of the FIFO may be in use simultaneously.

In the Framework logic, the FIFO interrupts are controlled to prevent false interrupts. The FIFO threshold value written into the FIFO interrupt control register is also used as a count value for the interrupt control. Once an interrupt is tripped by the threshold comparison, then the count value MUST be read/written to the FIFO before another interrupt will be signaled. Once the count value satisfied, then another interrupt will be signaled when the threshold is crossed.

IO Devices

Clock Generation PLLs

Quadia has two whereas Duet has only one software programmable PLL that may be used as communications, sample rate or FPGA clocks. The PLLs are very low jitter at less than 2.7 ps RMS and are stable to less than 10 ppm. This makes the PLL useful as a clock to the Rocket IO or as a sample rate clock for analog PMC modules, both applications that require low jitter.

PLL Device

The PLL device is an ICS8442 (http://www.icst.com/pdf/ics8442.pdf). The PLL device is capable of generating frequencies ranging from 30 to 300 MHz. The PLL is controlled over a three-wire serial port directly connected to the application FPGA which is controlled over the command channel in the FrameWork Logic. This allows the host PCI to control the PLL frequency configuration.

PLL Reference Input

The PLL may be driven by either a crystal or by an external frequency reference. The reference frequency must be in the range of 10 to 25 MHz. Naturally, the quality of the reference clock affects the PLL output quality so a stable clean clock is preferred.

The reference crystal for the PLL is 14.40 MHz, CTS part number KXN6489B (www.ctscorp.com). The crystal specifications are provided here.

Electrical Specifications:

KXN6489B	Specification
Nominal Frequency	14.4 Mhz
Mode of Vibration	Fundamental
Storage Temperature Range	-40 C to 85 C
Frequency Stability over Temperature	± 15 PPM (-10 C to 50C)
Frequency Make Tolerance	± 10 PPM @25 C +/- 3°C
Resonance Resistance	30Ω Max.
Drive Level	10 μ W Max.
Load Capacitance	12 pf
Shunt Capacitance	5.0 pf Max.
Aging	+/-2 PPM/Yr @25C

Figure 27. PLL Reference Crystal Specifications

An external reference clock may be used for Quadia using the J1 or J2 external input for PLL0 and PLL 1 respectively. (Requires a patch on Rev C cards). This external input directly drives the PLL Test Clock input and is selected using the internal clock multiplexer in the PLL. This input has a 33 ohm series resistor after the SMB input.

PLL Frequency Generation

The PLL has an internal VCO that is a multiple of the reference frequency and must be programmed to within an output range so that it locks. The VCO output frequency is

Fvco = Fref * M, 10 MHz < Fref <25 MHz

where Fref = 14.4 MHz for the PMC UWB on-card crystal, or is an external input and M is in the range of 1 to 511. Note that since Fref is limited to the range of 10 MHz to 25 MHz, external clock inputs should be constrained to this range by the input divisor.

Fvco must be in the range of

250 MHz < Fvco < 700 MHz

for the internal VCO to lock. Therefore, M must always be selected to meet this criteria.

The output of the PLL is then

Fpll = Fvco / N

where N = 1,2,4,8.

For the 14.4 MHz reference crystal on-board the module, this gives a range of 32.4 (M= 18, N = 8) to 691 MHz (M=48, N = 1).

The following table gives the PLL frequencies for values of M and N with valid outputs, when the frequency of the VCO, is less than 750 MHz and greater then 250 MHz.

PLL Connections
The output of the PLLs are LVDS and connects to the Cluster FPGAs. PLL0 connects to cluster0 FPGA, PLL1 connects to cluster1 FPGA. The pins are shown here. See the Developing Custom Logic chapter for more implementation details.

Signal	Cluster FPGA Pins
PLL Output +	AJ17
PLL Output -	AH17

Table 34. PLL Clock Connections to Cluster FPGAs

PLL Control Registers
These registers are used control the PLLs. There is one register for each PLL. Only PLL0 is available for Duet.

Ref Freq	14.4 MHZ				
		N			
M	FVCO (MHZ)	1	2	4	8
18	259.2	259.2	129.6	64.8	32.4
19	273.6	273.6	136.8	68.4	34.2
20	288	288	144.0	72.0	36.0
21	302.4	302.4	151.2	75.6	37.8
22	316.8	316.8	158.4	79.2	39.6
23	331.2	331.2	165.6	82.8	41.4
24	345.6	345.6	172.8	86.4	43.2
25	360	360	180.0	90.0	45.0
26	374.4	374.4	187.2	93.6	46.8
27	388.8	388.8	194.4	97.2	48.6
28	403.2	403.2	201.6	100.8	50.4
29	417.6	417.6	208.8	104.4	52.2
30	432	432	216.0	108.0	54.0
31	446.4	446.4	223.2	111.6	55.8
32	460.8	460.8	230.4	115.2	57.6
33	475.2	475.2	237.6	118.8	59.4
34	489.6	489.6	244.8	122.4	61.2
35	504	504	252.0	126.0	63.0
36	518.4	518.4	259.2	129.6	64.8
37	532.8	532.8	266.4	133.2	66.6
38	547.2	547.2	273.6	136.8	68.4
39	561.6	561.6	280.8	140.4	70.2
40	576	576	288.0	144.0	72.0
41	590.4	590.4	295.2	147.6	73.8
42	604.8	604.8	302.4	151.2	75.6
43	619.2	619.2	309.6	154.8	77.4
44	633.6	633.6	316.8	158.4	79.2
45	648	648	324.0	162.0	81.0
46	662.4	662.4	331.2	165.6	82.8
47	676.8	676.8	338.4	169.2	84.6
48	691.2	691.2	345.6	172.8	86.4

Bit	Function	Default State after Reset
0	PLL reset. This bit controls the PLL reset pin..	1 = reset PLL
1	PLL enable. This bit allows data to be shifted from the Velocia FPGA to the PLL.	0 = not enabled
2...31	Not Used	

Table 35. PLL Control Registers (write, PLL0 = +0x34, PLL1 = +0x35)

PLL Data Registers
These registers are used send the control numbers M,N and test pin select to the PLL. There is one register for each PLL.

Bit	Function	Value
8...0	PLL M coefficient. This is the M value that will be set in the PLL as a VCO multiplier factor.	0..511
10...9	PLL N coefficient. This is the N divisor that will be set in the PLL.	0 = Divide by 1 1 = Divide by 2 2 = Divide by 4 3 = Divide by 8
12...11	Test pin output select. Set to “00” for normal use.	
31...13	Not used	-

Table 36. PLL Data Registers (write, PLL0 = +0x28, PLL1 = +0x2C)

PLL Use
The PLL should be reset first by writing to the PLL Control Register. Check the Status Register and be sure that the PLL interface is not busy from a previous data write to either PLL (the PLLs share this physical interface so it could be busy). Then value for N,M should be loaded by writing to the PLL Data Register. Then the PLL should be released from reset by writing to the PLL Control Register. Resetting the device allows the PLL to lock on the new N,M values reliably.

PMC/XMC Modules

Note : XMC support has been added to Quadia beginning with Rev E.

Quadia and Duet supports two PMC/XMC module sites, one in each cluster, for IO expansion. The PMC/XMC may be used as a PCI device and may also use the J4 connector for a private data interface to the Virtex2 Pro logic for custom logic developments. A wide variety of modules are available from many vendors to support many types of IO requirements.

PMC Mechanicals
The PMC/XMC module sites conform to IEEE standard P1386.1 for mechanicals. Refer to this specification for all mechanical design. Innovative has layout and bracket design files that may be useful to designers. Contact technical support for access to these files.

Quadia has the optional J4 connector is populated and is keyed for 3.3V operation on the PCI bus. Double-wide modules are supported.

PMC PCI Interface
This PMC module site connects to the local PCI bus as 64-bit device. The local PCI bus supports 33 MHz and 3.3V signaling levels. Busmastering is supported from the PMCs. Maximum data rate over the baseboard local PCI bus to/from any PMC module is 264M bytes per second (64 bits at 33 MHz).

The four interrupts from the PMC module may be routed to the PCI bus or to a DSP. Routing the interrupt to a DSP allows baseboard to service PMC modules without involving the host resulting in better real-time performance. Custom drivers are required for this mode in many cases however.

PMC J4 Support
The PMC module site J4 connector can be used as a private data interface to the FPGA for custom logic designs. The J4 connector provides 64 connections to the FPGA, that may be used either as LVTTL connections, or as 32 pairs of LVDS signals. This allows the FPGA to be integrated into the data path of the PMC devices if high speed processing, formatting or analysis is required.

XMC Support
The XMC interface is composed of 4 lanes of 2G bit per second serial data operating over 8 differential signal pairs. These signal pairs are direct connections to the Xilinx Rocket IO ports on the cluster application logic. As such, the functionality of the XMC interface may be designed as part of the application logic. The serial pairs may be used individually, or as bonded sets, to achieve the desired bandwidth and data link functionality required by the application.

XMC Rocket IO Pairs

The XMC serial lanes are directly connected to Xilinx Rocket IO ports on the application logic through connector P15 (site 0) and P16 (site 1). These are differential signal pairs using CML (current mode logic). Each pair has a 50 ohm impedance on the PMC.

XMC Site	XMC Port	Application Logic Rocket IO Pair	Application Logic Signal Pins	P15/P16 XMC Connector Pins
0	Tx 0	14	AP8(+)/AP9(-)	D1(+)/E1(-)
0	Rx0	14	AP7(+)/AP6(-)	A1(+)/B1(-)
0	Tx 1	18	AP3(+)/AP2(-)	D3(+)/E3(-)
0	Rx1	18	AP4(+)/AP5(-)	A3(+)/B3(-)
0	Tx 2	21	A8(+)/A9(-)	D5(+)/E5(-)
0	Rx2	21	A7(+)/A6(-)	A5(+)/B5(-)
0	Tx 3	23	A16(+)/A17(-)	D7(+)/E7(-)
0	Rx3	23	A15(+)/A14(-)	A7(+)/B7(-)
1	Tx 0	16	AP8(+)/AP9(-)	D1(+)/E1(-)
1	Rx0	16	AP7(+)/AP6(-)	A1(+)/B1(-)
1	Tx 1	14	AP3(+)/AP2(-)	D3(+)/E3(-)
1	Rx1	14	AP4(+)/AP5(-)	A3(+)/B3(-)
1	Tx 2	7	A8(+)/A9(-)	D5(+)/E5(-)
1	Rx2	7	A7(+)/A6(-)	A5(+)/B5(-)
1	Tx 3	9	A16(+)/A17(-)	D7(+)/E7(-)
1	Rx3	9	A15(+)/A14(-)	A7(+)/B7(-)

Table 37. XMC Rocket IO Pairs

XMC Controls

The XMC interface has control signals that allow it to be used as the primary interface to the host, or provide additional functionality beyond the links. On the Velocia PMCs, these signals are connected directly to the application logic and have no functionality assigned to them in the standard logic. Here is a list of the signals and the typical use that can be implemented in custom application logic.

XMC Signal Name	P15/P16 XMC Connector Pin Number	Typical Signal Function
MSCL	F16	XMC ID ROM clock signal (I^2C interface)
MSDA	F14	XMC ID ROM data signal (I^2C interface)
GA0	F10	Geographic Address bit 0. Defines location of the module in the system.
GA1	C12	Geographic Address bit 1. Defines location of the module in the system.
GA2	C14	Geographic Address bit 2. Defines location of the module in the system.
MRSTI#	F2	XMC reset input, active low.
MBIST#	C11	XMC BIST (Built In Self-Test) input, active low.

Table 38. XMC Control and Support Signals

Custom XMC Applications

Custom applications using the XMC interface give the designer the capability of using the high speed data path for real-time signal processing applications. A variety of data protocols may be used, including completely custom applications implemented in the Application Logic.

Most custom applications may use the Rocket IO communications components provided in the FrameWork Logic design tools. A simple Rocket IO component, using a single serial lane, implements a point-to-point link that includes data buffering and flow control. For many signal processing applications, this provides the basic link from the PMC to the host. Velocia host applications may use the same components on both ends of the link, thus providing a data link from FPGA to FPGA. See the *FrameWork Logic User Guide* for information.

Custom applications that include custom host card should carefully follow the XMC and rocket IO specifications. Signal integrity of the 2 GHz signal pairs requires particular attention. The Xilinx Rocket IO User Guide provides helpful routing guidelines and signal integrity analysis information.

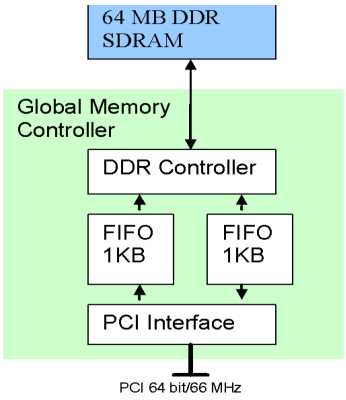
Memory Pool

Quadia has a 64M byte global memory pool residing on the secondary PCI bus. This memory pool is useful for applications that require common data for all DSPs or data sharing amongst the DSPs. Data can be put into the global memory pool so that it does not have to leave the card or interact with the host memory for data storage. Applications such as image processing benefit from this local memory pool because all the local PCI devices can access the data without burdening the host PCI bus.

Memory Pool Architecture

The memory pool controller is implemented in the Velocia logic. The controller has 1K bi-directional FIFOs for data buffering from the PCI bus. The controller allows the DDR SDRAM memory to be used by any PCI device as simple

memory. All of the DRAM control, including refresh, is handled automatically. From a user perspective this memory pool is simply a large memory residing on the local PCI bus.



The global memory pool enumerates as a 64 MB memory region on the Quadia.

A simplified view of the memory controller is shown here.

Figure 28. Simplified View of Memory Pool Controller

Memory Pool Performance
The memory pool controller allows full rate (264 MB/s on the 33 MHz, 64-bit local PCI bus) write bursts to memory up to the size of the 64MB size of the memory and read bursts of 256 to/from the PCI bus. The PCI bus may be 33 or 66 MHz, depending on the rate of the Quadia secondary PCI.

Memory Pool Use Rules
The memory pool controller handles all of the interface and control requirements of the DDR SDRAM devices used on Quadia. The DDR initialization and refresh are controlled by the logic.

During initialization, the controller will not be ready for approximately 300 uS after reset. This is to allow for the clocks to stabilize and the initialization sequence of the memory to be completed. DDR SDRAM memory has a specific initialization process that is automated by the logic and performed during this time. The initialization process is begun by toggling the DDR INITIALIZATION bit in the Quadia control register. The status register should then be monitored for the DCM (Virtex2 Pro clock PLLs) to indicate that they are stable and ready for use.

There are a few usage rules to follow for the memory pool

PCI Access Rules

1. All PCI access are for a burst of two 32-bit words minimum.

2. Access should start on 32-bit boundaries.

Memory Read Rules

1. Maximum burst size for reads is 512 32-bit words. After 512, the controller will disconnect. For reads larger than 512 32-bit words, the transfer will resume as expected at the next address. No special handling of this is required, this is just explained to help the user understand operation.

2. PCI access is terminated on row boundaries of the RAM. So if a read starts at any address in the RAM other than 512*n, n being a row number, the controller disconnects at the row end and resumes the next transaction at the start of the next row. No special handling of this is required, this is just explained to help the user understand operation.

3. The controller will issue retries to the PCI if a refresh is in progress. No special handling of this is required, this is just explained to help the user understand operation.

4. The controller will issue retries if the write FIFO is not empty. This preserves data coherency by guaranteeing that the write data is in memory before a read can be performed. No special handling of this is required, this is just explained to help the user understand operation.

5. The Read FIFO is flushed when a transaction terminates to insure data coherency. No special handling of this is required, this is just explained to help the user understand operation.

Memory Write Rules

1. Automatic precharge, bank change and refresh is performed after each write. No special handling of this is required, this is just explained to help the user understand operation.

2. The controller will issue retries to the PCI until the write FIFO is empty. This is to guarantee data coherency. No special handling of this is required, this is just explained to help the user understand operation.

3. If the write FIFO is full, the controller will disconnect from the PCI bus and issue retries. No special handling of this is required, this is just explained to help the user understand operation.

Software Support
Software methods are provided to control the memory pool initialization process and for use. Since the memory pool is seen as a simple 64MB memory device, the only special requirements for using are the word alignment on the addresses and the minimum access size. Support software in Pismo provides access control.

FPGAs

Quadia has three FPGAs: two application FPGAs (one for Duet) and the Velocia control FPGA. The Velocia FPGA provide core functionality controls to the card, while the FPGAs in each cluster are used for signal processing and application specific functions.

The cluster FPGA is intended for DSP applications and has a logic development system for both MATLAB and VHDL users. This system is described in detail in the *FrameWork Logic User Guide*. Many of the features implemented on Quadia and Duet are part of the FrameWork Logic such as the data plane communications. Users can build on top of this infrastructure

to incorporate DSP functions and system features using logic development system. Developers should see the *FrameWork Logic User Guide* for more information.

Velocia FPGA
The Velocia FPGA provides card support functions such as the memory pool controller, reset controls, FPGA SelectMap interface and PLL interface. A provision for Rocket IO links to the cluster FPGAs has been provided to give PCI access to the Cluster FPGAs. No user development is normally done on the Velocia FPGA. Contact technical support if modifications to the functionality are required.

The Velocia FPGA is a Xilinx Virtex2 Pro, XC2VP4-5FG456C.

Reprogramming the Velocia FPGA
Updates may be released as required to fix bugs or enhance the Velocia FPGA functionality. A Motorola Hex (.exo) file is provided for these updates that is the bit image for the Velocia FPGA.

Updating the Velocia FPGA is done using the VelociaVSPROM application provided with the development tools. After starting the program, browse to the new .exo file and select it. The reprogramming will take less than a minute to complete. After the Velocia FGPA image is reprogrammed, the computer must be power-cycled to load the new image.

! Do not turn off the machine if the programming fails to complete for any reason. Attempt to reburn the device. Failure to complete the process will require factory service to reburn the ROM!

Cluster FPGA Devices
Each cluster has a large application FPGA for signal processing and custom functionality. The standard device is the Xilinx XC2VP40. In addition to the resources in the summary table below, Quadia adds external memory and integrates the FPGA in tightly with the other peripherals on the card.

Other devices are available as special orders for size or device speed.

Device	Logic Cells	Embedded 18Kb Memory Blocks	Multipliers	PowerPC Cores
XC2VP30	30816	136	136	2
XC2VP40-5FF1152C (Standard)	43632	192	192	2
XC2VP50	53136	232	232	2

Table 39. Summary of Compatible Cluster FPGA Devices

Cluster FPGA Connections

The cluster FPGAs are integrated in the baseboard with tight coupling to the cluster DSPs, IO devices, local memory, and the other FPGA. In this chapter, we will acquaint you with the connections and what is implemented in the Framework Logic. Customizing the interfaces and logic design is described in the *FrameWork Logic User Guide*.

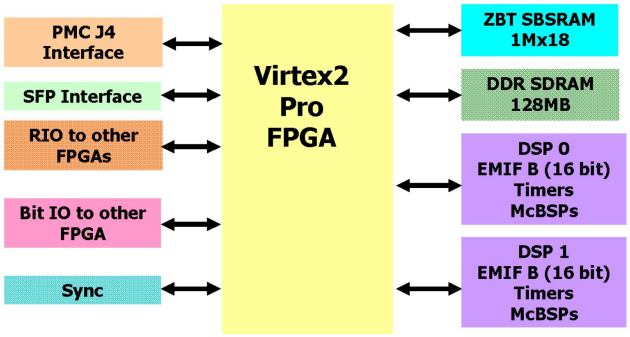

PMC J4 Interface
SFP Interface
RIO to other FPGAs
Bit IO to other FPGA
Sync
Virtex2 Pro FPGA
ZBT SBSRAM 1Mx18
DDR SDRAM 128MB
DSP 0 EMIF B (16 bit) Timers McBSPs
DSP 1 EMIF B (16 bit) Timers McBSPs

Figure 29. Simplified View of Cluster FPGA Connections

FPGA DSP Connections

The cluster FPGA has the two DSPs in its cluster connected to the logic. EMIF B data bus is used exclusively for the connection and provides a 133 MHz, 16-bit data path to the DSPs. The Framework logic has a memory map as described in this chapter that supports the peripherals in that design. All of the EMIF control signals are connected to the FPGA, plus address lines BEA16 through BEA20 for custom designs. The maximum data rate through EMIF B connection is 266 MB per second. Efficient interfaces in the Framework logic demonstrate the use of FIFOs to deliver high rate data to the DSP. More advanced designs may also take advantage of Peripheral Data Transfer (PDT) supported by the DSP to maximize the rate of data transfers from the FPGA to DSP devices.

In addition to the data bus connection, the DSP MCBSP ports are also connected to the FPGA. The McBSP ports provide a convenient serial port interface between the DSP that complements the data bus connection. The McBSP ports are described fully in the TI C6000 DSP peripherals document (SPRU190). The connection to the cluster FPGAs is a straightforward hookup that may be used in custom logic designs. There is no specific use of the serial ports in the Framework logic.

There are four interrupts from each DSP, INT4 through INT7, plus four GP connections (GP1-GP4). The GP pins may be used as DMA interrupts or as general purpose IO. In the Framework logic, the interrupts have specific assignments as described in the DSP interrupts section of this chapter.

The DSP timer 0 is also connected to the FPGA. There is no specific use of the DSP timer in the Framework logic.

Cluster FPGA Memory

The application FPGA has an SBSRAM device and a DDR SDRAM attached to it for use by FPGA applications. These memories provide data buffering and computational RAM for FPGA applications.

The SRAM device connected to each cluster FPGA is a 2 MB, organized as 1M by 18 bits. This device is a Cypress CY7C1372 which supports clock rates up to 200 MHz. The Framework Logic provides a simple test and demonstration

interface for The SBSRAM that is described in detail in the *FrameWork Logic User Guide*. The underlying controller is an implementation of Xilinx Application Note XAPP163. FPGA logic developers can easily replace the simple register interface logic to build on top of the high performance logic core when integrating the SRAM into their logic design.

The DDR SDRAM memory device attached to each cluster FPGA is a 32MB device, organized as 16M by 16 bits (Micron part MT46V16M16TG-75). This device is capable of up to 133 MHz data bus. In the Framework logic, the DDR SDRAM use is demonstrated as a very large FIFO memory. The logic component, referred to as VFIFO, controls the DDR SDRAM to create the FIFO functionality. VFIFO controls the read and write memory pointers and moves data from a small input FIFO into the memory device and retrieving data to a small FIFO so that from the user perspective it is just a large FIFO. Custom logic applications can use the VFIFO component as is, or can modify it to use the SDRAM in other buffer applications.

The FPGA memories may also be used by the PowerPC cores in custom applications. In this case, the logic developer should refer to Xilinx libraries for memory controllers and connection methods to the PowerPC bus.

Cluster FPGA Miscellaneous Connections

There are 16 uncommitted connections from FPGA to FPGA. These signals may be used in custom logic designs for any purpose - triggers, controls between FPGAs, and status reporting are just a few uses. These signals are not used in the Framework Logic.

Cluster FPGA Power Supplies

The cluster FPGAs are powered by on-card power supplies that generate the FPGA core voltage and IO voltages. The heaviest load is usually on the core power supply. Custom FPGA designs should be run through Xilinx XPower tool to estimate the power required. Keep in mind that transient loading can be higher instantaneously, but steady state values must be met by these local supplies.

Voltage	Use	Maximum Load
1.5V	FPGA Core Voltages	15A total
2.5V	2.5V VCCO and aux voltage	3A total (SBSRAM and DDR chips consume up to 1.5A of this)
1.25V	Vref for SSTL banks and terminations	1.5A (terminations consume 0.5A of this)
3.3V	3.3V VCCOs	Directly sourced from the host

Table 40. FPGA Power Supplies

Loading the Cluster FPGA image

There are two methods of loading the cluster FPGA image: JTAG or SelectMap via PCI. The cluster FPGA image must be loaded after each power-on as there is no local storage of the image.

During FPGA development, the JTAG interface is a convenient method for working with the cluster FPGAs for download and debug. A single JTAG chain connects the two FPGAs in series (cluster 0 then cluster 1 in the chain) and connects to standard Xilinx download cables such as Parallel Cable IV using JP3 and JP14. The JTAG chain on JP3 has the cluster 0 FPGA and the Velocia control FPGA; JP14 has cluster 1 FPGA and the FLASH device used for the Velocia FPGA. The

Pinouts for JP3 and JP14 and connector description are in the appendix of this manual. JTAG clock speeds of up to 10 MHz have been used.

Note: The cluster FPGAs are both connected to the JTAG chain on JP3 for board revisions A through E. When Impact is started, the JTAG chain will be automatically scanned and two VP40 devices will be identified. The first device is cluster 0 FPGA, the second is cluster 1 FPGA. For each device a .bit file must either be assigned or the device must be bypassed. Impact can then program the devices and should support a successful configuration.

For image downloads, Xilinx Impact software tool is commonly used. When Impact is started, the JTAG chain is scanned and each device is identified. A bit file should be assigned to the FPGA, the other device should be bypassed. DO NOT PROGRAM THE VELOCIA FPGA OR THE VELOCA FPGA FLASH. THIS WILL PREVENT THE CARD FROM OPERATING PROPERLY.

Connector	Device 0	DQuadiaevice 1
JP3	Velocia FPGA	Cluster 0 FPGA
JP14	Velocia FLASH	Cluster 1 FPGA

Table 41. Quadia JTAG Chains and Connectors

Duet Connector	Device 0	Device 1
JP2	Velocia FPGA	Cluster FPGA

Table 42. Duet JTAG Chains and Connectors

Other Xilinx tools, such as ChipScope and System Generator, use the JTAG for debug and development. The same general steps are followed to first establish the scan path and identify the devices in the chain. Once this is achieved, the tools are ready for use. If the JTAG scan fails for any reason, try powering down everything and restarting. The tools will not work unless the scan path is working.

! Do not hot-plug the JTAG connector to the FPGAs. Damage may occur!

The FPGA is loaded over the SelectMap interface from the PCI bus. An application, VelociaLoader, is provided that allows FPGA images to be downloaded to the cluster FPGAs. The loader program reads a Motorola S-Record formatted image (.exo) and programs the FPGA over the PCI bus to its SelectMap interface. The loading process takes about 1 minute to complete for each FPGA. The two cluster FPGA images are loaded separately.

The Framework logic is provided as an EXO formatted logic file. Custom logic files can use the same tool to download images after an EXO is generated. The technique for making these files is described in the developing custom logic chapter of this manual.

External IO Input to Cluster FPGAs

Quadia has an external IO signal on J1 and J2, an SMB coaxial input connector, that may be used as an input or output. These signals have a 33 ohm series terminator on them. The input connector is 50 ohm SMB. These signals are LVTTL compatible (0 <0.7 v, 1 > 2.4v) but are NOT 5V tolerant.

! Inputs should be limited to 3.3V or damage may occur!

Custom logic designs can use these signals for a variety of purposes for clocks, input triggers, or output signals. The FrameWork Logic does not implement any function on these pins. The signal is attached to FPGA pin H17.

Rear Terminal IO (Revisions D and above)

In addition to J1 and J2, Quadia has user IO from each application FPGA connected to user-defined rear terminal port J5. On Quadia, each application FPGA has 44 connections to connector CJ4 (which is located in PICMG connector space J5) that may be used in custom logic applications for things such as triggering, communications and debug. The signals are arranged as 22 differential signal pairs that may also be used single-ended.

Pinout for CJ4 and signal names are shown in the connector chapter in this document. Signal pairs are TP0/TP1, TP2/3 and so on for signal s TP0 through TP31. These differential pairs may be used as LVDS by changing the logic UCF (constraints) file.

Rear terminal connection to CJ4 use PICMG standard rear terminal cards. Since this is an undefined connector, users can implement any digital IO required by their application with a custom rear terminal IO card. More advanced applications can also use these signals to implement special signaling on the backplane of the system by using custom rear terminal cards. The rear terminal form-factor and card requirements are given in PICMG Compact PCI specification 2.0.

Duet PXI Support

Duet has support for PXI system features including local buses, triggers, system clock, and star clock. All of these signals are direct connections to the FPGA allowing logic designs to used these features for system triggering, communications and clocking. The *FrameWork Logic User Guide* provides additional information concerning the connections to the logic. No PXI functionality is implemented at this time in the FrameWork Logic for Duet.

PXI Signal	Signal Name	Function
Local Bus Left	PXI_LBL[12..0]	A 13-bit bus to the card in the slot left of the Duet
Local Bus Right	PXI_LBR[12..0]	A 13-bit bus to the card in the slot right of the Duet
Star Trigger	PXI_Star	A trigger or clock signal, usually to Duet, from the host that is not a shared trace.
Trigger	PXI_Trig[7..0]	Triggers in the PXI system. These are bidirectional and Duet many drive them.
10 Mhz Reference Clock	PXI_10MHz	A 10 Mhz reference clock in PXI systems into Duet.

Table 43. Duet PXI Signals

The PXI signals are connected to CJ2, the second compact PCI connector. The signals are level-shifted using zero-delay buffers so that PXI signals are 5V tolerant. The connectors chapter of this manual gives the signal connections to CJ2 and the FPGA.

Chapter 12. *Connector Pinouts and Physical Information*

The standard pad for pin 1 is a square pad on all Innovative Integration products. Connector pin numbering varies from part to part.

There are two sections to this chapter: Quadia and Duet. Each details the connectors and physical layout of the boards.

Quadia Connectors

PMC Private IO Connector (JN4)

The PMC JN4 connector is used to interface PMC modules directly to the Cluster 0 FPGA as a private data path.

Connector Types:	1mm double row, IEEE 1386 compatible vertical connector
Number of Connections:	64
Mating Connector:	Molex P/N 71436
Baseboard	Quadia

Table 44. PMC JN4

Pin	Function
1..64	PMC IO1..64

The PMC IO pins may be used as LVTTL (3.3V only) or as LVDS pairs. LVDS pairs are 1-2, 3-4, 5-6....63-64

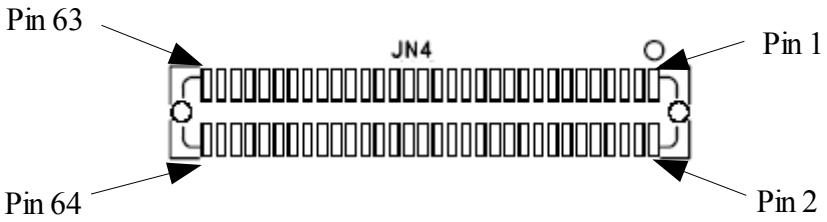


Figure 30. PMC JN4 Connector Pin out

PMC IO Connector (JN8)

The PMC JN8 connector is used to interface PMC modules directly to the Cluster 1 FPGA as a private data path. It also provides connection to Ethernet pairs on a 2.16 backplane when .

Connector Types:	1mm double row, IEEE 1386 compatible vertical connector
Number of Connections:	64
Mating Connector:	Molex P/N 71436
Baseboard	Quadia

Table 45. PMC JN8

Pin	Function
1	Ethernet Port 0 A+ / PMC1_IO1
2	Ethernet Port 0 C+ / PMC1_IO2
3	Ethernet Port 0 A- / PMC1_IO3
4	Ethernet Port 0 C- / PMC1_IO4
5	PMC1_IO5 / DGND (when R283 zero ohms is installed)
6	PMC1_IO6 / DGND (when R284 zero ohms is installed)
7	Ethernet Port 0 B+ / PMC1_IO7

Pin	Function
8	Ethernet Port 0 D+ / PMC1_IO8
9	Ethernet Port 0 B- / PMC1_IO9
10	Ethernet Port 0 D- / PMC1_IO10
11	PMC1_IO11 / DGND (when R280 zero ohms is installed)
12	PMC1_IO12 / DGND (when R285 zero ohms is installed)
13	Ethernet Port 1 A+ / PMC1_IO13
14	Ethernet Port 1 C+ / PMC1_IO14
15	Ethernet Port 1 A- / PMC1_IO15
16	Ethernet Port 1 C- / PMC1_IO16
17	PMC1_IO17 / DGND (when R281 zero ohms is installed)
18	PMC1_IO18 / DGND (when R282 zero ohms is installed)
19	Ethernet Port 1 B+ / PMC1_IO19
20	Ethernet Port 1 D+ / PMC1_IO20
21	Ethernet Port 1 B- / PMC1_IO21
22	Ethernet Port 1 D- / PMC1_IO22
23	PMC1_IO25 / DGND (when R30 zero ohms is installed)
24	PMC1_IO24 / DGND (when R286 zero ohms is installed)
25..64	PMC IO25..64

The PMC IO pins may be used as LVTTL (3.3V only) or as LVDS pairs. All signals are LVDS pairs from the logic IO1-2. IO3-4 and so on.

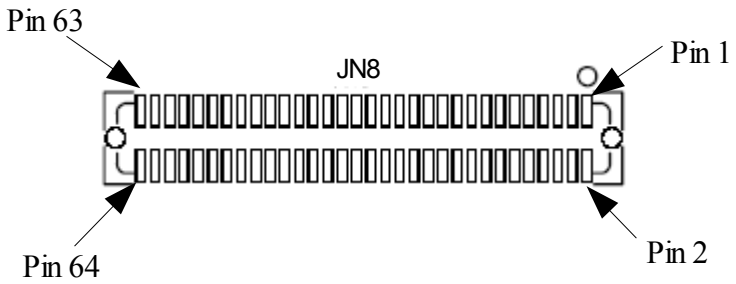


Figure 31. PMC JN8Connector Pin out

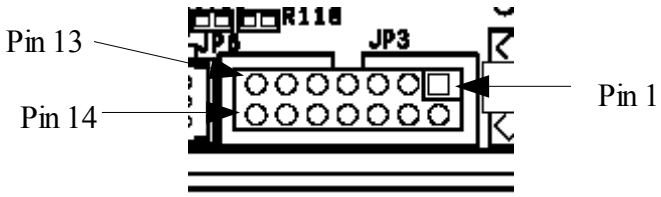
JP3 – FPGA JTAG Connector

Connector Types:	14 pin 2mm double row male header
Number of Connections:	14
Mating Connector:	AMP P/N 111623-3
Baseboard	Quadia

The following table gives the pin numbers and functions for the FPGA JTAG (JP3) connector. On Quadia and Duet, this JTAG connector is for Cluster 0 Virtex2 Pro and the Velocia control FPGA.

Table 46. FPGA JTAG Connector Pinouts

Pin Number	JP3 Function	Direction (from Quadia)
1,3,5,7,9,11,13	Ground	Power Return
2	+3.3V	Power
4	TMS	I
6	TCK	I
8	TDO	O
10	TDI	I
12,14	No Connect	-



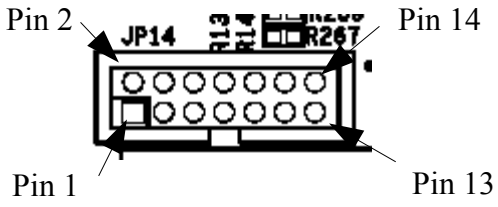
JP14 – FPGA JTAG Connector

Connector Types:	14 pin 2mm double row male header
Number of Connections:	14
Mating Connector:	AMP P/N 111623-3
Baseboard	Quadia

The following table gives the pin numbers and functions for JP14. On Quadia, this JTAG connector is for the Cluster 1 Virtex2 Pro FPGA and FLASH. On Duet, this connector is only for the FLASH.

Table 47. JTAG Connector for Velocia FPGA and FLASH

Pin Number	JP14 Function	Direction (from Quadia)
1,3,5,7,9,11,13	Ground	Power Return
2	+3.3V	Power
4	TMS	I
6	TCK	I
8	TDO	O
10	TDI	I
12,14	No Connect	-



JP1 – DSP JTAG

This connector is the JTAG scan path connection to the DSPs. The debug tools for the DSP plug into JP1 during development work.

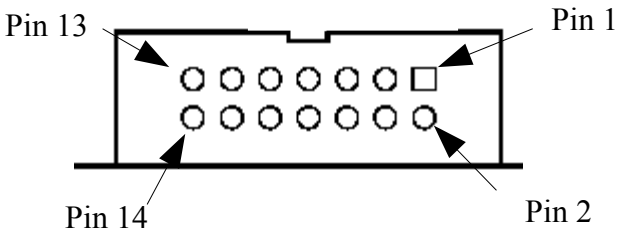
Connector Types:	14 pin 0.1” double row shrouded male header center polarized, pin 6 removed
Number of Connections:	14
Mating Connector:	AMP P/N 746285-2
Baseboard	Quadia and Duet

The following table gives the pin numbers and functions for the JP1 connector.

Table 48. JP1 DSP JTAG Connector Pinouts

Pin Number	JP1 Function	Direction (from Quadia)
1	TMS	I

Pin Number	JP1 Function	Direction (from Quadia)
2	TRST*	I
3	TDI	I
7	TDO	O
9, 11	TCK	I
13	EMU0	I
14	EMU1	I
5	+3.3V	O
4, 8, 10, 12	Digital Ground	Power



JP10 – Power Input Connector (Test Only)

Connector Types:	6 pin locking power connector (Molex 43045-0602)
Number of Connections:	6
Mating Connector:	Molex 43025-0600 and contacts
Baseboard	Quadia

The following table gives the pin numbers and functions for the JP10 connector. This connector is used for factory test.

Table 49. JP10 Power Input Connector (Test Only)

Pin Number	JP10 Function	Direction (from Quadia)
1	Digital +5V	Power
2,4	Digital ground	Power
3,5	No connect	-
6	3.3V	Power

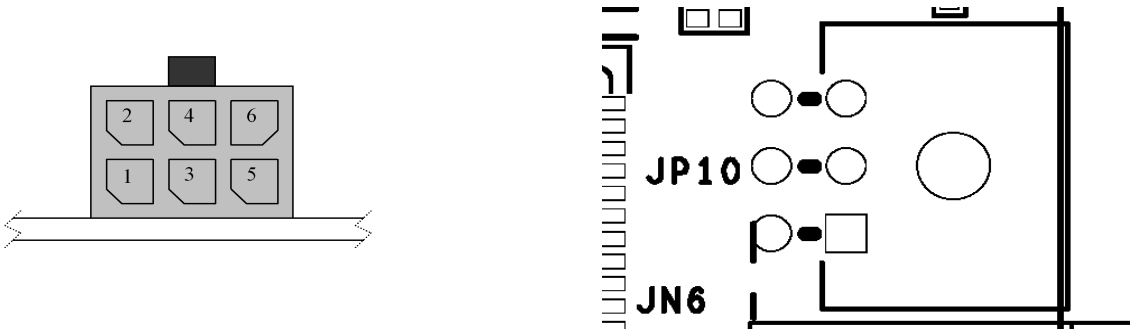


Figure 32. Power Connector Pin Positions (side view, from front of connector, showing connector keying and locking tab along with printed circuit board position)

Note: Mating connector may be numbered differently.

JE1, JE2 – Logic Testpoint Connectors (Quadia Rev C only)

Connector Types:	MICTOR - Impedance Controlled micropitch (AMP 767054-1)
Number of Connections:	44
Mating Connector:	AMP 767087-1 and others
Baseboard	Quadia only

The following table gives the pin numbers and functions for the JE1 and JE2 connector. The connectors have identical pinouts, but JE1 is for Cluster 0 FPGA and JE2 is for Cluster 1 FPGA This mates directly with many logic analyzer test probes.

JE1 : Cluster 0 FPGA

JE2 : Cluster 1 FPGA

Table 50. FPGA testpoint Connector

Pin Number	JE1, JE2 Function	Direction (from Quadia)
1,2,3,4	DGND	Power
5,6	Not used	-
7	FPGA testpoint 31	O
8	FPGA testpoint 30	O
9	FPGA testpoint 29	O
10	FPGA testpoint 28	O
11	FPGA testpoint 27	O
12	FPGA testpoint 26	O
13	FPGA testpoint 25	O
14	FPGA testpoint 24	O
15	FPGA testpoint 23	O
16	FPGA testpoint 22	O
17	FPGA testpoint 21	O
18	FPGA testpoint 20	O
19	FPGA testpoint 19	O
20	FPGA testpoint 18	O
21	FPGA testpoint 17	O
22	FPGA testpoint 16	O
23	FPGA testpoint 15	O
24	FPGA testpoint 14	O
25	FPGA testpoint 13	O
26	FPGA testpoint 12	O
27	FPGA testpoint 11	O
28	FPGA testpoint 10	O

Pin Number	JE1, JE2 Function	Direction (from Quadia)
29	FPGA testpoint 9	O
30	FPGA testpoint 8	O
31	FPGA testpoint 7	O
32	FPGA testpoint 6	O
33	FPGA testpoint 5	O
34	FPGA testpoint 4	O
35	FPGA testpoint 3	O
36	FPGA testpoint 2	O
37	FPGA testpoint 1	O
38	FPGA testpoint 0	O
39..43	DGND	Power
44	No Connect	-

External IO J1/J2

Connector Types:	SMB 50 Ohm
Mating Connector:	AMP P/N 413985-3 (straight) or AMP P/N 414002-7 (right angle)
Baseboard	Quadia

These connectors, J1 and J2, provide a connection to each cluster FPGA 0 and 1 respectively. They may be programmed to be input or output in each FPGA design.

PCI-X Enable JP15

Connector Types:	Header, 2mm, 2 pins
Mating Connector:	-
Baseboard	Quadia only

Short pin 1-2 to disable PCI-X compatibility .

Sync Connector JP2 (Rev C only)

Connector Types:	MDR, 26 position
Mating Connector:	3M 10126-6000EC (IDC wiremount) or 10126-3000VE

This connector provides access to the 12 differential pairs, 6 pairs from each cluster FPGA, that may be used for trigger synchronization or other purposes. They may be programmed to be input or output in each FPGA design.

In this table the the negative pin is Nx and positive pin is Px of the pair x.

Pin Number	JP2 Function	Direction (from Quadia)
1	DGND	Power
2	Sync N4	I/O
3	Sync P4	I/O
4	Sync N3	I/O
5	Sync P3	I/O
6	Sync N1	I/O
7	Sync P1	I/O
8	Sync N0	I/O
9	Sync P0	I/O

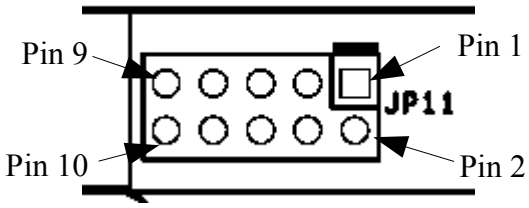
Pin Number	JP2 Function	Direction (from Quadia)
10	Sync N5	I/O
11	Sync P5	I/O
12	Sync N2	I/O
13	Sync P2	I/O
14	No Connect	-
15	Sync P11	I/O
16	Sync N11	I/O
17	Sync P6	I/O
18	Sync N6	I/O
19	Sync P7	I/O
20	Sync N7	I/O
21	Sync P9	I/O
22	Sync N9	I/O
23	Sync P10	I/O
24	Sync N10	I/O
25	Sync P8	I/O
26	Sync N8	I/O

JP11 – Factory Power Test Connector

Connector Types:	Shrouded header, pin 6 removed for key
Number of Connections:	10
Mating Connector:	AMP 746285-1
Baseboard	Quadia

This connector is used in factory test for measuring on-card voltages. These connections do NOT support the high currents required by the card during operation and are intended for measurement only.

Pin Number	JP11 Function	Direction (from Quadia)
1	DGND	Power
2	+5V supply	Power
3	DSP Core Voltage (1.2V for T processors, 1.4V for older devices)	Power
4	DDR Memory termination (1.25V)	Power
5	No Connect	-
6	PCI bridge core voltage (1.3V)	Power
7	DDR and FPGA supply (2.5V)	Power
8	FPGA Core Voltage (1.5V)	Power
9	3.3V	Power
10	No Connect	-



JP7, JP8 – SFP Connectors

Connector Types:	Shielded SFP module connector (AMP 1367073-1)
Number of Connections:	20
Mating Connector:	SFP Modules (see discussion for compatible modules)
Baseboard	Quadia

These SFP connectors allow industry-standard SFP modules to be used. JP7 is cluster 0 FPGA and JP8 is cluster 1 FPGA port. These devices are a direct connect to the Cluster FPGAs.

Pin Number	JP7/JP8 Function	Direction
(from SFP Device)		
1	DGND	Power
2	TX Fault	O
3	Disable	I
4	Serial Data	I/O
5	Serial Clock	I
6	Detect	O
7	Rate Select	I
8	Loss of Signal	O
9	DGND	Power
10	DGND	Power
11	DGND	Power
12	RX -	O
13	RX +	O
14	DGND	Power
15	VCC Rx (filtered 3.3V)	Power
16	VCC Tx (filtered 3.3V)	Power
17	DGND	Power
18	TX +	I
19	TX -	I
20	DGND	Power

JN1, JN5 – PMC Connectors

Connector Types:	1mm double row, IEEE 1386 compatible vertical connector
Number of Connections:	64
Mating Connector:	Molex P/N 71436
Quadia	

This is one of the PMC connectors, primarily used for the local PCI bus connection. JN1 is PMC 0; JN5 is PMC 1. JN5 is not applicable for Duet.

Table 51. JN1, JN5 – PMC Connectors

Pin Number	JN1/JN5 Function	Direction (from Quadia)
1	JTAG TCK	O
2	-12V	Power
3	DGND	Power
4	INTA#	I
5	INTB#	I
6	INTC#	I
7	Busmode 0	I
8	+5V	Power
9	INTD#	I
10	No connect	-
11	DGND	Power
12	System Mgmt Bus 3.3V	Power
13	PCI CLK	O
14	DGND	Power
15	DGND	Power
16	GNT 2#	O
17	REQ 2#	I
18	+5V	Power

Pin Number	JN1/JN5 Function	Direction (from Quadia)
19	+3.3V	Power
20	AD31	IO
21	AD28	IO
22	AD27	IO
23	AD25	IO
24	DGND	Power
25	DGND	Power
26	CBE3#	IO
27	AD22	IO
28	AD21	IO
29	AD19	IO
30	+5V	Power
31	+3.3V	Power
32	AD17	IO
33	Frame#	IO
34	DGND	Power
35	DGND	Power
36	IRDY#	IO
37	DEVSEL#	IO
38	+5V	Power
39	DGND	Power
40	LOCK#	IO
41	System Mgmt Bus Clk	O
42	System Mgmt Bus Data	IO
43	PAR	IO
44	DGND	Power
45	+3.3V	Power
46	AD15	IO
47	AD12	IO
48	AD11	IO

Pin Number	JN1/JN5 Function	Direction (from Quadia)
49	AD9	IO
50	+5V	Power
51	DGND	Power
52	CBE0#	IO
53	AD6	IO
54	AD5	IO
55	AD4	IO
56	DGND	Power
57	+3.3V	Power
58	AD3	IO
59	AD2	IO
60	AD1	IO
61	AD0	IO
62	+5V	Power
63	DGND	Power
64	REQ64#	IO

JN2, JN6 – PMC Connectors

Connector Types:	1mm double row, IEEE 1386 compatible vertical connector
Number of Connections:	64
Mating Connector:	Molex P/N 71436

This is one of the PMC connectors, primarily used for the local PCI bus connection. JN2 is PMC 0; JN6 is PMC 1. JN6 is not applicable for Duet.

Reserved1 through Reserved 8 connect to the cluster FPGA.

Pin Number	JN2/JN6 Function	Direction (from Quadia)
1	+12V	Power
2	JTAG TRST#	O
3	TMS	O
4	TDO	I
5	TDI	O
6	DGND	Power
7	DGND	Power
8	No connect	-
9	No connect	-
10	No connect	-
11	+3.3V	Power
12	+3.3V	Power
13	Reset#	O
14	DGND	Power
15	+3.3V	Power
16	DGND	Power
17	No connect	-
18	DGND	Power
19	AD30	IO
20	AD29	IO
21	DGND	Power
22	AD25	IO
23	AD24	IO
24	+3.3V	Power
25	IDSEL	O
26	AD23	IO
27	+3.3V	Power
28	AD20	IO
29	AD18	IO
30	DGND	Power

Pin Number	JN2/JN6 Function	Direction (from Quadia)
31	AD16	Power
32	CBE2#	IO
33	DGND	Power
34	Reserved3	IO
35	TRDY#	IO
36	+3.3V	Power
37	DGND	Power
38	STOP#	IO
39	PERR	IO
40	DGND	Power
41	+3.3V	Power
42	SERR	IO
43	CBE1#	IO
44	DGND	Power
45	AD14	IO
46	AD13	IO
47	M66EN	I
48	AD10	IO
49	AD8	IO
50	+3.3V	Power
51	AD7	IO
52	Reserved4	IO
53	+3.3V	Power
54	Reserved5	IO
55	Reserved1	IO
56	DGND	Power
57	Reserved2	IO
58	Reserved6	IO
59	DGND	Power
60	Reserved7	IO

Pin Number	JN2/JN6 Function	Direction (from Quadia)
61	ACK64#	IO
62	+3.3V	Power
63	DGND	Power
64	Reserved8	IO

JN3, JN7 – PMC Connectors

Connector Types:	1mm double row, IEEE 1386 compatible vertical connector
Number of Connections:	64
Mating Connector:	Molex P/N 71436

This is one of the PMC connectors, primarily used for the local PCI bus connection. JN3 is PMC 0; JN7 is PMC 1. JN7 is not applicable for Duet.

Pin Number	JN3/JN7 Function	Direction (from Quadia)
1	No Connect	-
2	DGND	Power

Pin Number	JN3/JN7 Function	Direction (from Quadia)
3	DGND	Power
4	CBE7#	IO
5	CBE6#	IO
6	CBE5#	IO
7	CBE4#	IO
8	DGND	Power
9	+3.3V	Power
10	PAR64	IO
11	AD63	IO
12	AD62	IO
13	AD61	IO
14	DGND	Power
15	DGND	Power
16	AD60	IO
17	AD59	IO
18	AD58	IO
19	AD57	IO
20	DGND	Power
21	DGND	Power
22	AD56	IO
23	AD55	IO
24	AD54	IO
25	AD53	IO
26	DGND	Power
27	DGND	Power
28	AD52	IO
29	AD51	IO
30	AD50	IO
31	AD49	IO
32	DGND	Power

Pin Number	JN3/JN7 Function	Direction (from Quadia)
33	DGND	Power
34	AD48	IO
35	AD47	IO
36	AD46	IO
37	AD45	IO
38	DGND	Power
39	+3.3V	Power
40	AD44	IO
41	AD43	IO
42	AD42	IO
43	AD41	IO
44	DGND	Power
45	DGND	Power
46	AD40	IO
47	AD39	IO
48	AD38	IO
49	AD37	IO
50	DGND	Power
51	DGND	Power
52	AD36	IO
53	AD35	IO
54	AD34	IO
55	AD33	IO
56	DGND	Power
57	+3.3V	Power
58	AD32	IO
59	No Connect	-
60	No Connect	-
61	No Connect	-
62	DGND	Power

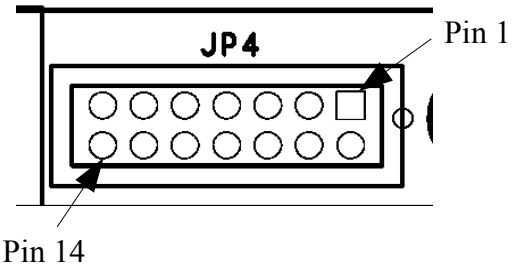
Pin Number	JN3/JN7 Function	Direction (from Quadia)
63	DGND	Power
64	No Connect	-

JP4 – JTAG Test Connector for PCI Bridges and Miscellaneous

Connector Types:	2 mm double row header
Number of Connections:	14
Mating Connector:	AMP P/N 111623-3
Baseboard	Quadia and Duet

The following table gives the pin numbers and functions for the JP4, the JTAG connector for the PCI bridges and others. This connector is used for factory test only.

Pin Number	JP4 Function	Direction (from Quadia)
1,3,5,7,9,11,13	DGND	Power
2	+3.3V	Power
4	TMS	I
6	TCK	I
8	TDO	O
10	TDI	I
12	No Connect	-
14	TRST#	I



JP5 – Velocia V2Pro FPGA test points Connector

Connector Types:	0.1 in single row header (usually depopulated)
Number of Connections:	6
Mating Connector:	-

The following table gives the pin numbers and functions for the JP5, the Velocia FPGA test point probe connector. This connector is used for factory test only.

Pin Number	JP5 Function	Direction (from Quadia)
1..6	Test Point 0..5	IO

CJ3 – StarFabric and Ethernet Backplane Connector

Connector Types:	2 mm hard metric, AMP 352171-1
Number of Connections:	95, including shield connections
Mating Connector:	AMP P/N 1-352272-1
Baseboard	Quadia Only

The following table gives the pin numbers and functions for the CJ3, the backplane connector for StarFabric(PCIMG 2.17) and Ethernet (PCIMG 2.16).

NOTE : StarFabric is only available on Quadia Rev C.

Pin Number	CJ3 Function	Direction (from Quadia)
A1	DGND	Power
A2	StarFabric Link 0 TX 0+ (Rev C only)	O
A3	StarFabric Link 0 TX 1+ (Rev C only)	O
A4	StarFabric Link 0 TX 2+ (Rev C only)	O
A5	StarFabric Link 0 TX 3+ (Rev C only)	O
A6	StarFabric Link 1 TX 0+ (Rev C only)	O
A7	StarFabric Link 1 TX 1+ (Rev C only)	O
A8	StarFabric Link 2 TX 2+ (Rev C only)	O
A9	StarFabric Link 2 TX 3+ (Rev C only)	O
A10	DGND	Power
A11	No Connect	-
A12	No Connect	-
A13	No Connect	-
A14	DGND	Power
A15	Ethernet LP 1 DB+	IO
A16	Ethernet LP 1 DA+	IO
A17	Ethernet LP 0 DB+	IO
A18	Ethernet LP 0 DA+	IO
A19	System Geographical Addr 4	I
B1	DGND	Power
B2	StarFabric Link 0 TX 0- (Rev C only)	O
B3	StarFabric Link 0 TX 1- (Rev C only)	O
B4	StarFabric Link 0 TX 2- (Rev C only)	O
B5	StarFabric Link 0 TX 3- (Rev C only)	O
B6	StarFabric Link 1 TX 0- (Rev C only)	O
B7	StarFabric Link 1 TX 1- (Rev C only)	O

Pin Number	CJ3 Function	Direction (from Quadia)
B8	StarFabric Link 2 TX 2- (Rev C only)	O
B9	StarFabric Link 2 TX 3- (Rev C only)	O
B10	DGND	Power
B11	No Connect	-
B12	No Connect	-
B13	No Connect	-
B14	DGND	Power
B15	Ethernet LP 1 DB-	IO
B16	Ethernet LP 1 DA-	IO
B17	Ethernet LP 0 DB-	IO

Pin Number	CJ3 Function	Direction (from Quadia)
B18	Ethernet LP 0 DA-	IO
B19	System Geographical Addr 3	I
C1..C10	DGND	Power
C11..C18	No Connect	-
C19	System Geographical Addr 2	I
D1	DGND	Power
D2	StarFabric Link 0 RX 0+ (Rev C only)	I
D3	StarFabric Link 0 RX 1+ (Rev C only)	I
D4	StarFabric Link 0 RX 2+ (Rev C only)	I
D5	StarFabric Link 0 RX 3+ (Rev C only)	I
D6	StarFabric Link 1 RX 0+ (Rev C only)	I
D7	StarFabric Link 1 RX 1+ (Rev C only)	I
D8	StarFabric Link 2 RX 2+ (Rev C only)	I
D9	StarFabric Link 2 RX 3+ (Rev C only)	I
D10	DGND	Power
D11	No Connect	-
D12	No Connect	-
D13	No Connect	-
D14	DGND	Power
D15	Ethernet LP 1 DD+	IO
D16	Ethernet LP 1 DC+	IO
D17	Ethernet LP 0 DD+	IO
D18	Ethernet LP 0 DC+	IO
D19	System Geographical Addr 1	I
E1	DGND	Power
E2	StarFabric Link 0 RX 0- (Rev C only)	I
E3	StarFabric Link 0 RX 1- (Rev C only)	I
E4	StarFabric Link 0 RX 2- (Rev C only)	I
E5	StarFabric Link 0 RX 3- (Rev C only)	I

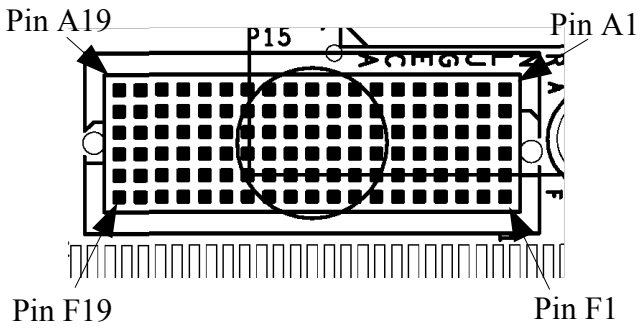
Pin Number	CJ3 Function	Direction (from Quadia)
E6	StarFabric Link 1 RX 0- (Rev C only)	I
E7	StarFabric Link 1 RX 1- (Rev C only)	I
E8	StarFabric Link 2 RX 2- (Rev C only)	I
E9	StarFabric Link 2 RX 3- (Rev C only)	I
E10	DGND	Power
E11	No Connect	-
E12	No Connect	-
E13	No Connect	-
E14	DGND	Power
E15	Ethernet LP 1 DD-	IO
E16	Ethernet LP 1 DC-	IO
E17	Ethernet LP 0 DD-	IO
E18	Ethernet LP 0 DC-	IO
E19	System Geographical Addr 0	I
Z1..Z19	DGND	Power

P15, P16 – XMC Connectors (Quadia Rev E + only)

Connector Types:	XMC connector, Samtec ASP-105884-01
Number of Connections:	114, arranged as 6 rows of 19 pins each
Mating Connector:	Samtec ASP-105885-01
Baseboard	Quadia (P15, P16)

P15 and P16 are XMC connectors to the XMC modules. These are compatible with VITA 42.0 standard. P15 is XMC for site 0, P16 is XMC for site 1.

Pin	Signal	Direction
D1	Rx Pair 0 +	I
E1	Rx Pair 0 +	I
A3	Tx Pair 1 +	O
B3	Tx Pair 1 +	O
D3	Rx Pair 1 +	I
E3	Rx Pair 1 +	I
A5	Tx Pair 2 +	O
B5	Tx Pair 2 +	O
D5	Rx Pair 2 +	I
E5	Rx Pair 2+	I
A7	Tx Pair 3 +	O
B7	Tx Pair 3 +	O
D7	Rx Pair 3 +	I
E7	Rx Pair 3 +	I
F16	MSCL (XMC ID ROM clock)	I
F14	MSDA (XMC ID ROM data)	I/O
F10	GA0 (XMC geographic address 0)	I
C12	GA1 (XMC geographic address 1)	I
C14	GA2 (XMC geographic address 2)	I
F2	MRSTI# (XMC reset, active low)	I
C11	BIST	I
A2,A4,A6,A8,A10,A12,A14,A18, B2,B4,B6,B8,B10,B12, B14,B16,B18,C2,C4,C6,C8,C10,C12,C14,C16,C18,D2,D4,D6,D8,D10,D12,D14,D16,D18, E2,E4,E6,E8,E10,E12, E14,E16,E18	Digital Ground	Power
C1,C3,C5,C7	3.3V from host	Power
All others	No Connect	-



CJ5 - Compact PCI Rear Terminal User IO (Quadia Rev E + only)

Connector Types:	2mm hard metric, Framatone HM2R70PA5108N9
Number of Connections:	110, arranged as 5 rows of 22 pins each
Mating Connector:	AMP 352131-1
Baseboard	Quadia

CJ5 is the compact PCI rear terminal IO connector that provides direct connection to the application FPGAs. Signal pairs are TP0/1, TP2/3 and so on.

CJ5 Pin Number	Function	FPGA	FPGA Pin Number	Direction (from Quadia)
A1	C1_TP3	1	AE22	I/O
A2	C1_TP2	1	AL23	I/O
A3	C1_TP7	1	AJ22	I/O
A4	C1_TP6	1	AH22	I/O
A5	C1_TP16	1	AL21	I/O
A6	C1_TP21	1	AD19	I/O
A7	C1_TP18	1	AF20	I/O
A9	SYNC_0+	0	AH14	I/O
A10	SYNC_1+	0	AJ15	I/O
A11	SYNC_2+	0	AJ16	I/O
A12	SYNC_3+	0	AK11	I/O

CJ5 Pin Number	Function	FPGA	FPGA Pin Number	Direction (from Quadia)
A13	SYNC_4+	0	AL12	I/O
A14	SYNC_5+	0	AM13	I/O
A15	C0_TP31	0	AG18	I/O
A16	C0_TP19	0	AH20	I/O
A17	C0_TP16	0	AL21	I/O
A18	C0_TP12	0	AG21	I/O
A19	C0_TP20	0	AJ20	I/O
A20	C0_TP2	0	AL23	I/O
A21	C0_TP7	0	AJ22	I/O
A22	C0_TP5	0	AG22	I/O
B1	C1_TP5	1	AG22	I/O
B2	C1_TP4	1	AF22	I/O
B3	C1_TP10	1	AE21	I/O
B4	C1_TP11	1	AF21	I/O
B5	C1_TP17	1	AE20	I/O
B6	C1_TP20	1	AJ20	I/O
B7	C1_TP19	1	AH20	I/O
B8	C1_TP31	1	AG18	I/O
B9	SYNC_0-	0	AG14	I/O
B10	SYNC_1-	0	AH15	I/O
B11	SYNC_2-	0	AH16	I/O
B12	SYNC_3-	0	AJ11	I/O
B13	SYNC_4-	0	AL13	I/O
B14	SYNC_5-	0	AM14	I/O
B15	C0_TP30	0	AF18	I/O
B16	C0_TP18	0	AF20	I/O
B17	C0_TP17	0	AE20	I/O
B18	C0_TP13	0	AH21	I/O

CJ5 Pin Number	Function	FPGA	FPGA Pin Number	Direction (from Quadia)
B19	C0_TP21	0	AD19	I/O
B20	C0_TP3	0	AE22	I/O
B21	C0_TP6	0	AH22	I/O
B22	C0_TP4	0	AF22	I/O
C1..C22	DGND	-	-	POWER
D1	C1_TP29	1	AE18	I/O
D2	C1_TP28	1	AD18	I/O
D3	C1_TP22	1	AE19	I/O
D4	C1_TP23	1	AF19	I/O
D5	C1_TP13	1	AH21	I/O
D6	C1_TP12	1	AG21	I/O
D7	C1_TP8	1	AK22	I/O
D8	C1_TP0	1	AK24	I/O
D9	SYNC_6+	1	AH14	I/O
D10	SYNC_7+	1	AJ15	I/O
D11	SYNC_8+	1	AJ16	I/O
D12	SYNC_9+	1	AK11	I/O
D13	SYNC_10+	1	AL12	I/O
D14	SYNC_11+	1	AM13	I/O
D15	C0_TP10	0	AE21	I/O
D16	C0_TP28	0	AD18	I/O
D17	C0_TP22	0	AE19	I/O
D18	C0_TP24	0	AG19	I/O
D19	C0_TP26	0	AJ19	I/O
D20	C0_TP14	0	AJ21	I/O
D21	C0_TP8	0	AK22	I/O
D22	C0_TP0	0	AK24	I/O

CJ5 Pin Number	Function	FPGA	FPGA Pin Number	Direction (from Quadia)
E1	C1_TP24	1	AG19	I/O
E2	C1_TP25	1	AH19	I/O
E3	C1_TP26	1	AJ19	I/O
E4	C1_TP27	1	AK19	I/O
E5	C1_TP14	1	AJ21	I/O
E6	C1_TP15	1	AK21	I/O
E7	C1_TP9	1	AL22	I/O
E8	C1_TP1	1	AL24	I/O
E9	SYNC_6-	1	AG14	I/O
E10	SYNC_7-	1	AH15	I/O
E11	SYNC_8-	1	AH16	I/O
E12	SYNC_9-	1	AJ11	I/O
E13	SYNC_10-	1	AL13	I/O
E14	SYNC_11-	1	AM14	I/O
E15	C0_TP11	0	AF21	I/O
E16	C0_TP29	0	AE18	I/O
E17	C0_TP23	0	AF19	I/O
E18	C0_TP25	0	AH19	I/O
E19	C0_TP27	0	AK19	I/O
E20	C0_TP15	0	AK21	I/O
E21	C0_TP9	0	AL22	I/O
E22	C0_TP1	0	AL24	I/O
F1..F22	DGND	-	-	POWER

Board Layouts

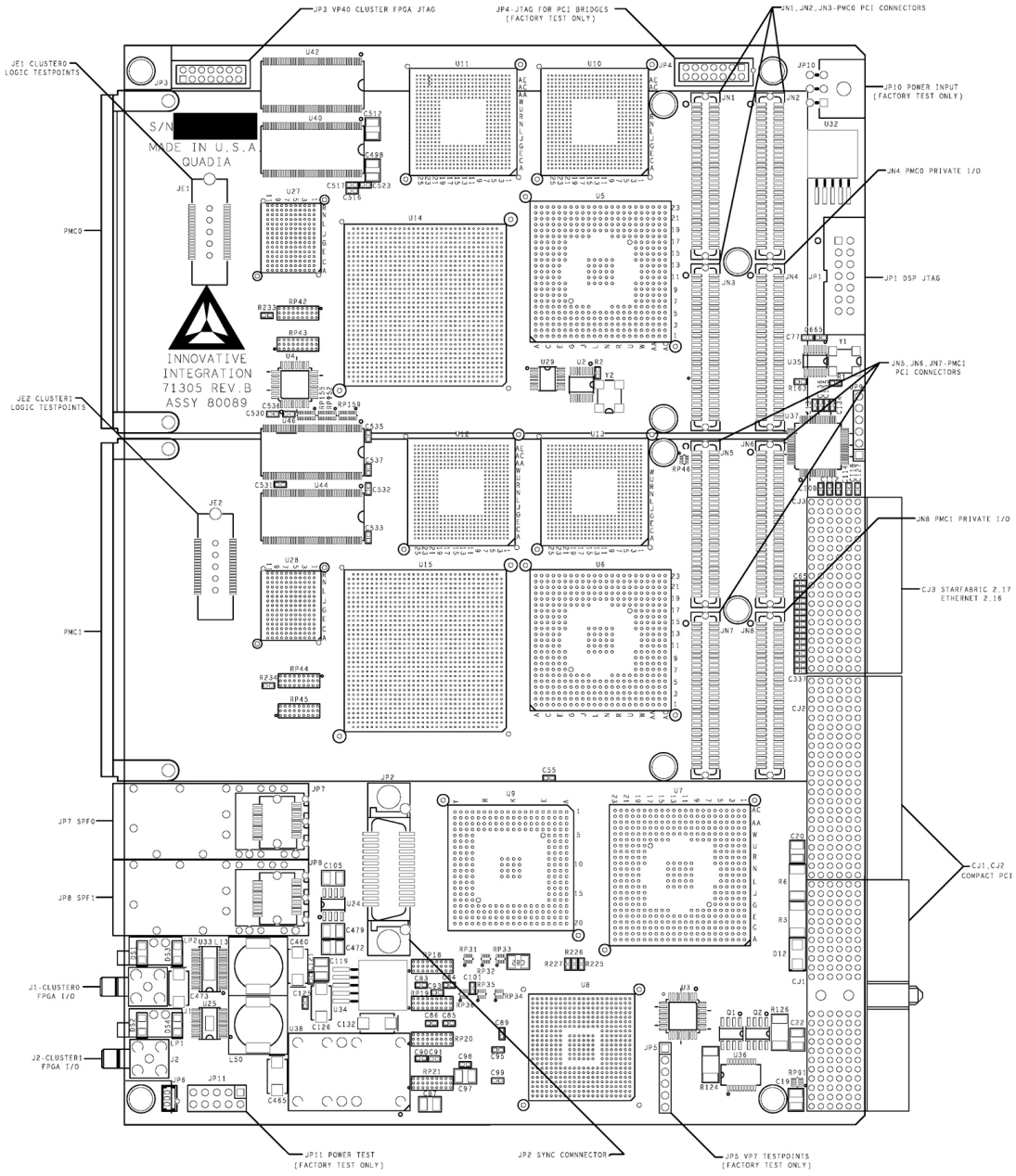


Figure 33. Quadia Rev C Board Layout

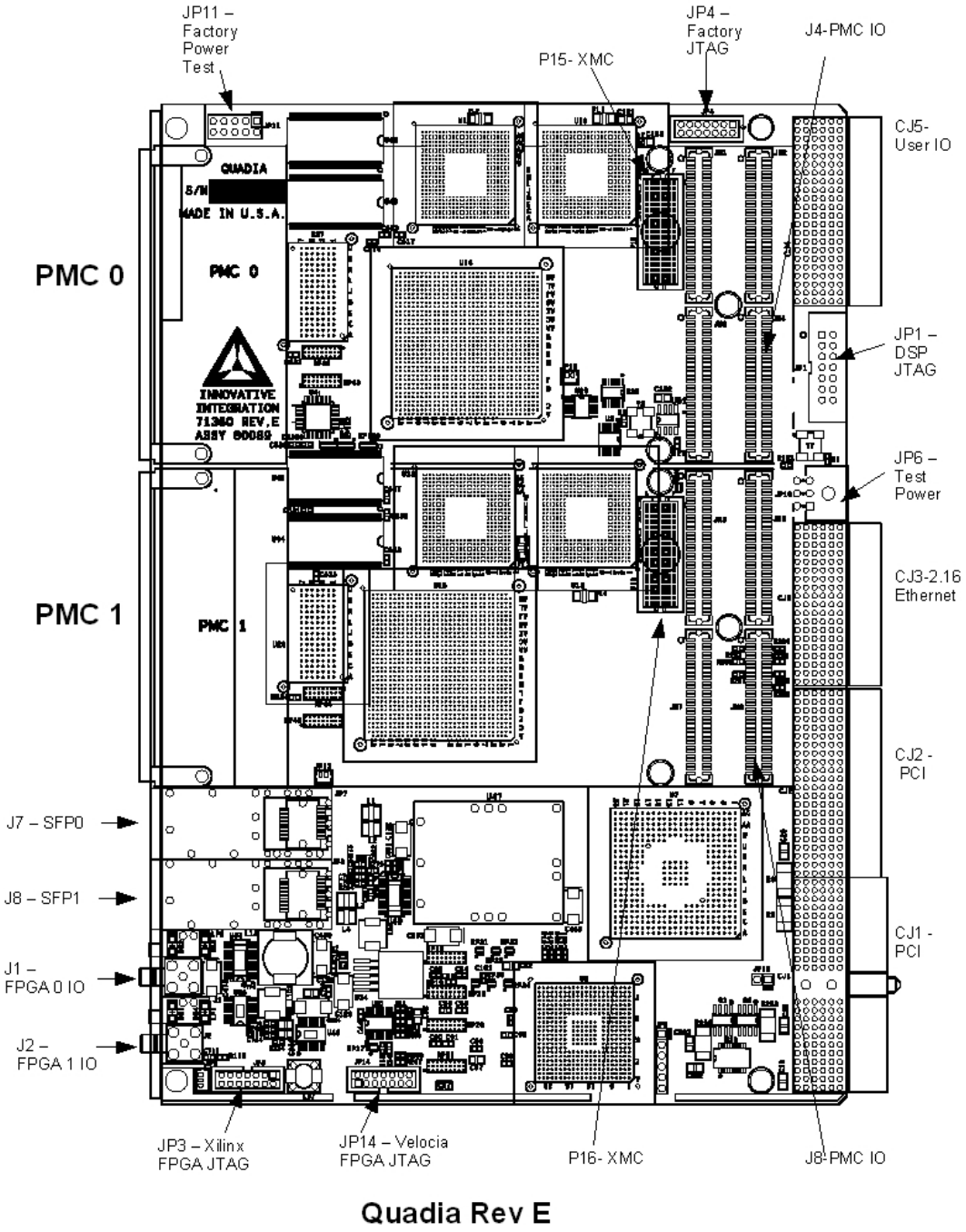


Figure 34. Quadia Rev D/E Board Layout

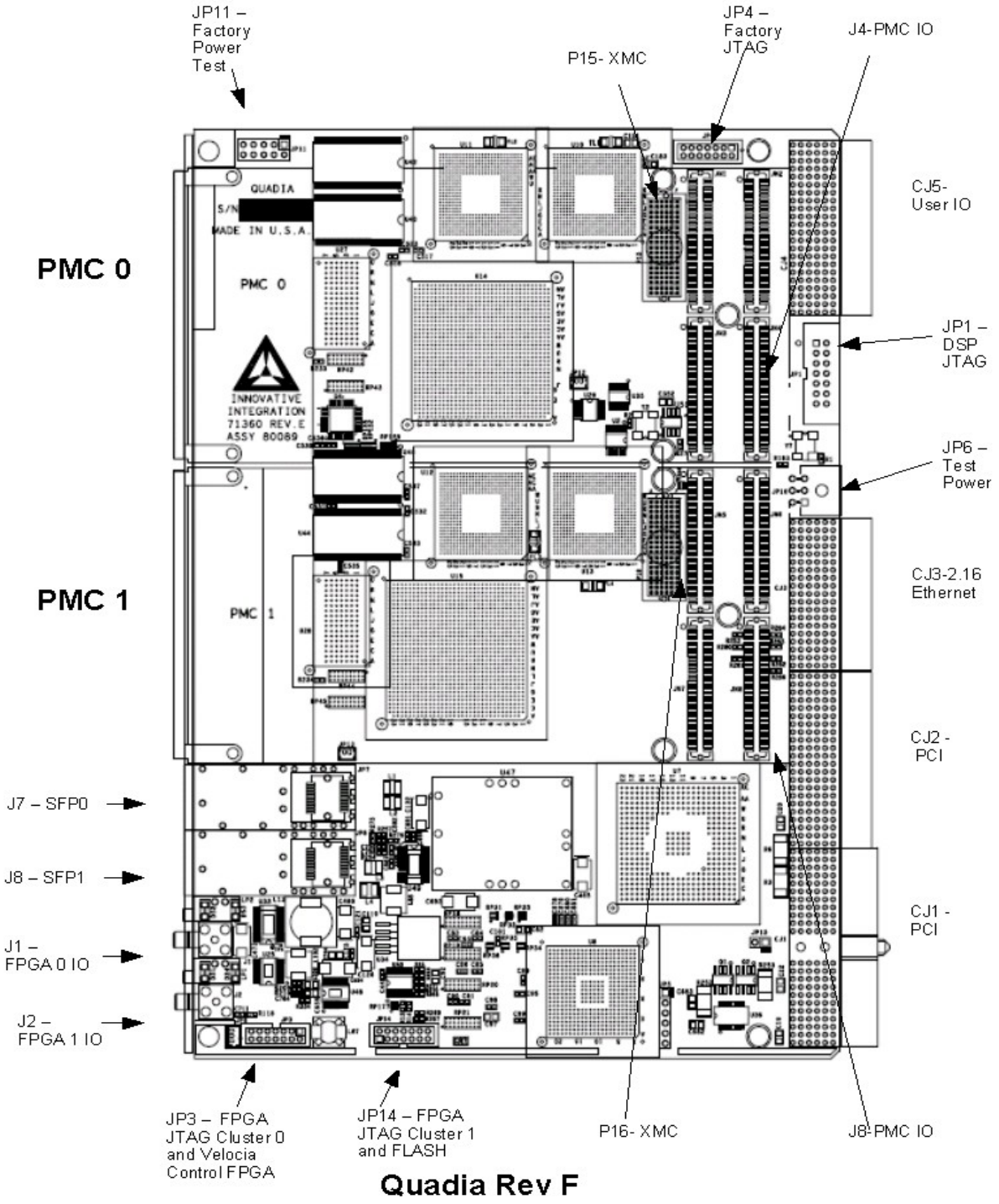


Figure 35. Quadia Rev F Board Layout

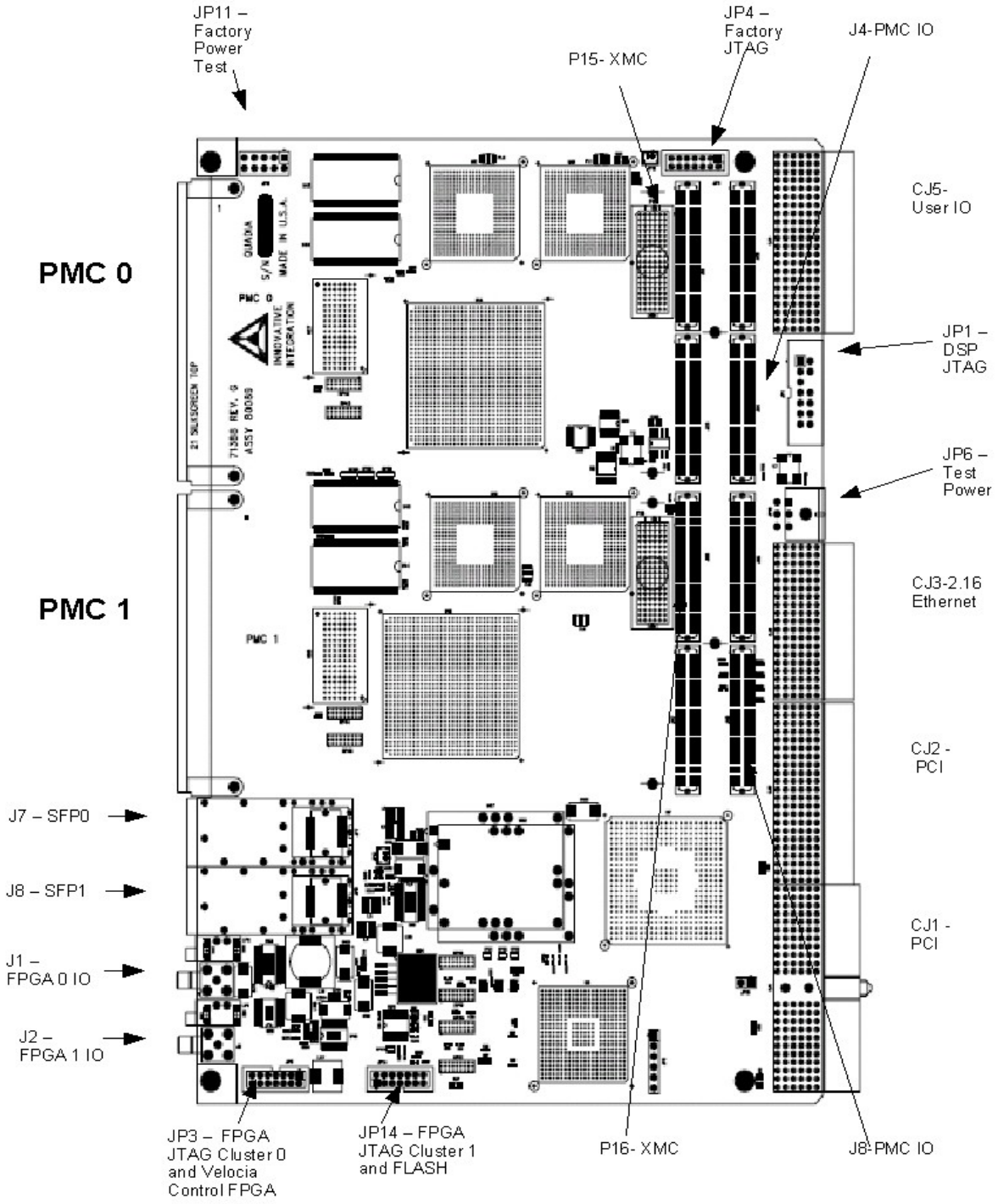


Figure 36. Quadia Rev G Board Layout

Duet Connectors

PMC Private IO Connector (JN4)

The PMC JN4 connector is used to interface PMC module directly to the Duet Cluster FPGA as a private data path.

Connector Types:	1mm double row, IEEE 1386 compatible vertical connector
Number of Connections:	64
Mating Connector:	Molex P/N 71436
Baseboard	Duet

Table 52. PMC JN4

Pin	Function
1..64	PMC IO1..64

The PMC IO pins may be used as LVTTL (3.3V only) or as LVDS pairs. LVDS pairs are 1-2, 3-4, 5-6....63-64

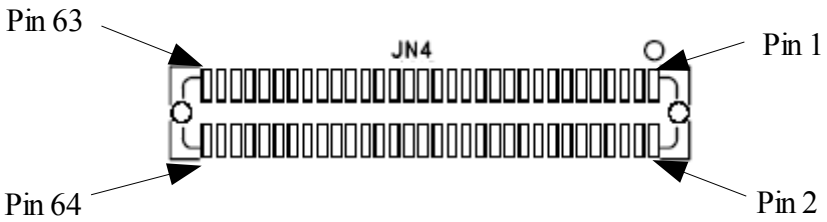


Figure 37. PMC JN4 Connector Pin out

JP2 – FPGA JTAG Connector

Connector Types:	14 pin 2mm double row male header
Number of Connections:	14
Mating Connector:	AMP P/N 111623-3
Baseboard	Duet

The following table gives the pin numbers and functions for the FPGA JTAG (JP2) connector. On Duet, this JTAG connector is for the Virtex2 Pro and the Velocia control FPGA.

Table 53. FPGA JTAG Connector Pinouts

Pin Number	JP2 Function	Direction (from Duet)
1,3,5,7,9,11,13	Ground	Power Return
2	+3.3V	Power
4	TMS	I
6	TCK	I
8	TDO	O
10	TDI	I
12,14	No Connect	-

JP6 – Velocia FLASH JTAG Connector

Connector Types:	14 pin 2mm double row male header
Number of Connections:	14
Mating Connector:	AMP P/N 111623-3
Baseboard	Duet

The following table gives the pin numbers and functions for JP6. On Duet, this connector is only for the FLASH.

Table 54. JTAG Connector for Velocia FLASH

Pin Number	JP6 Function	Direction (from Duet)
1,3,5,7,9,11,13	Ground	Power Return
2	+3.3V	Power
4	TMS	I
6	TCK	I
8	TDO	O
10	TDI	I
12,14	No Connect	-

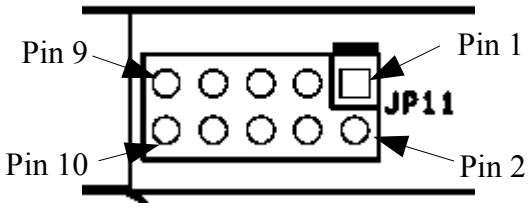
JP4 – Factory Power Test Connector

Connector Types:	Shrouded header, pin 6 removed for key
Number of Connections:	10
Mating Connector:	AMP 746285-1
Baseboard	Duet

This connector is used in factory test for measuring on-card voltages. These connections do NOT support the high currents required by the card during operation and are intended for measurement only.

Pin Number	JP4 Function	Direction (from Quadia)
1	DGND	Power
2	+5V supply	Power
3	DSP Core Voltage (1.2V for T processors, 1.4V for older devices)	Power
4	DDR Memory termination (1.25V)	Power
5	No Connect	-
6	PCI bridge core voltage (1.3V)	Power
7	DDR and FPGA supply (2.5V)	Power

Pin Number	JP4 Function	Direction (from Quadia)
8	FPGA Core Voltage (1.5V)	Power
9	3.3V	Power
10	No Connect	-



JP1 – DSP JTAG

This connector is the JTAG scan path connection to the DSPs. The debug tools for the DSP plug into JP1 during development work.

Connector Types:	14 pin 0.1” double row shrouded male header center polarized, pin 6 removed
Number of Connections:	14
Mating Connector:	AMP P/N 746285-2
Baseboard	Duet

The following table gives the pin numbers and functions for the JP1 connector.

Table 55. JP1 DSP JTAG Connector Pinouts

Pin Number	JP1 Function	Direction (from Quadia)
1	TMS	I
2	TRST*	I
3	TDI	I

Pin Number	JP1 Function	Direction (from Quadia)
7	TDO	O
9, 11	TCK	I
13	EMU0	I
14	EMU1	I
5	+3.3V	O
4, 8, 10, 12	Digital Ground	Power

JN1 – PMC Connector

Connector Types:	1mm double row, IEEE 1386 compatible vertical connector
Number of Connections:	64
Mating Connector:	Molex P/N 71436
Baseboard	Duet

This is one of the PMC connectors, primarily used for the local PCI bus connection.

Table 56. JN1 – PMC Connectors

Pin Number	JN1 Function	Direction (from Duet)
1	JTAG TCK	O
2	-12V	Power
3	DGND	Power
4	INTA#	I
5	INTB#	I
6	INTC#	I
7	Busmode 0	I
8	+5V	Power
9	INTD#	I
10	No connect	-

Pin Number	JN1 Function	Direction (from Duet)
11	DGND	Power
12	System Mgmt Bus 3.3V	Power
13	PCI CLK	O
14	DGND	Power
15	DGND	Power
16	GNT 2#	O
17	REQ 2#	I
18	+5V	Power
19	+3.3V	Power
20	AD31	IO
21	AD28	IO
22	AD27	IO
23	AD25	IO
24	DGND	Power
25	DGND	Power
26	CBE3#	IO
27	AD22	IO
28	AD21	IO
29	AD19	IO
30	+5V	Power
31	+3.3V	Power
32	AD17	IO
33	Frame#	IO
34	DGND	Power
35	DGND	Power
36	IRDY#	IO
37	DEVSEL#	IO
38	+5V	Power
39	DGND	Power
40	LOCK#	IO

Pin Number	JN1 Function	Direction (from Duet)
41	System Mgmt Bus Clk	O
42	System Mgmt Bus Data	IO
43	PAR	IO
44	DGND	Power
45	+3.3V	Power
46	AD15	IO
47	AD12	IO
48	AD11	IO
49	AD9	IO
50	+5V	Power
51	DGND	Power
52	CBE0#	IO
53	AD6	IO
54	AD5	IO
55	AD4	IO
56	DGND	Power
57	+3.3V	Power
58	AD3	IO
59	AD2	IO
60	AD1	IO
61	AD0	IO
62	+5V	Power
63	DGND	Power
64	REQ64#	IO

JN2 – PMC Connectors

Connector Types:	1mm double row, IEEE 1386 compatible vertical connector
Number of Connections:	64
Mating Connector:	Molex P/N 71436
Baseboard	Duet

This is one of the PMC connectors, primarily used for the local PCI bus connection.

Reserved1 through Reserved 8 connect to the cluster FPGA.

Pin Number	JN2 Function	Direction (from Duet)
1	+12V	Power
2	JTAG TRST#	O
3	TMS	O
4	TDO	I
5	TDI	O
6	DGND	Power
7	DGND	Power
8	No connect	-
9	No connect	-
10	No connect	-
11	+3.3V	Power
12	+3.3V	Power
13	Reset#	O
14	DGND	Power
15	+3.3V	Power
16	DGND	Power
17	No connect	-
18	DGND	Power
19	AD30	IO

Pin Number	JN2 Function	Direction (from Duet)
20	AD29	IO
21	DGND	Power
22	AD25	IO
23	AD24	IO
24	+3.3V	Power
25	IDSEL	O
26	AD23	IO
27	+3.3V	Power
28	AD20	IO
29	AD18	IO
30	DGND	Power
31	AD16	Power
32	CBE2#	IO
33	DGND	Power
34	Reserved3	IO
35	TRDY#	IO
36	+3.3V	Power
37	DGND	Power
38	STOP#	IO
39	PERR	IO
40	DGND	Power
41	+3.3V	Power
42	SERR	IO
43	CBE1#	IO
44	DGND	Power
45	AD14	IO
46	AD13	IO
47	M66EN	I
48	AD10	IO
49	AD8	IO

Pin Number	JN2 Function	Direction (from Duet)
50	+3.3V	Power
51	AD7	IO
52	Reserved4	IO
53	+3.3V	Power
54	Reserved5	IO
55	Reserved1	IO
56	DGND	Power
57	Reserved2	IO
58	Reserved6	IO
59	DGND	Power
60	Reserved7	IO
61	ACK64#	IO
62	+3.3V	Power
63	DGND	Power
64	Reserved8	IO

JN3 – PMC Connector

Connector Types:	1mm double row, IEEE 1386 compatible vertical connector
Number of Connections:	64
Mating Connector:	Molex P/N 71436
Baseboard	Duet

This is one of the PMC connectors, primarily used for the local PCI bus connection.

Pin Number	JN3 Function	Direction (from Duet)
1	No Connect	-

Pin Number	JN3 Function	Direction (from Duet)
2	DGND	Power
3	DGND	Power
4	CBE7#	IO
5	CBE6#	IO
6	CBE5#	IO
7	CBE4#	IO
8	DGND	Power
9	+3.3V	Power
10	PAR64	IO
11	AD63	IO
12	AD62	IO
13	AD61	IO
14	DGND	Power
15	DGND	Power
16	AD60	IO
17	AD59	IO
18	AD58	IO
19	AD57	IO
20	DGND	Power
21	DGND	Power
22	AD56	IO
23	AD55	IO
24	AD54	IO
25	AD53	IO
26	DGND	Power
27	DGND	Power
28	AD52	IO
29	AD51	IO
30	AD50	IO
31	AD49	IO

Pin Number	JN3 Function	Direction (from Duet)
32	DGND	Power
33	DGND	Power
34	AD48	IO
35	AD47	IO
36	AD46	IO
37	AD45	IO
38	DGND	Power
39	+3.3V	Power
40	AD44	IO
41	AD43	IO
42	AD42	IO
43	AD41	IO
44	DGND	Power
45	DGND	Power
46	AD40	IO
47	AD39	IO
48	AD38	IO
49	AD37	IO
50	DGND	Power
51	DGND	Power
52	AD36	IO
53	AD35	IO
54	AD34	IO
55	AD33	IO
56	DGND	Power
57	+3.3V	Power
58	AD32	IO
59	No Connect	-
60	No Connect	-
61	No Connect	-

Pin Number	JN3 Function	Direction (from Duet)
62	DGND	Power
63	DGND	Power
64	No Connect	-

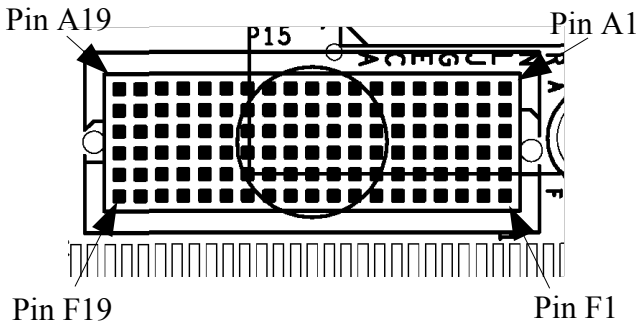
P1 – XMC Connectors

Connector Types:	XMC connector, Samtec ASP-105884-01
Number of Connections:	114, arranged as 6 rows of 19 pins each
Mating Connector:	Samtec ASP-105885-01
Baseboard	Duet

P1 is the XMC connector to the PMC/XMC module. These are compatible with VITA 42.0 standard.

Pin	Signal	Direction
D1	Rx Pair 0 +	I
E1	Rx Pair 0 +	I
A3	Tx Pair 1 +	O
B3	Tx Pair 1 +	O
D3	Rx Pair 1 +	I
E3	Rx Pair 1 +	I
A5	Tx Pair 2 +	O
B5	Tx Pair 2 +	O
D5	Rx Pair 2 +	I
E5	Rx Pair 2+	I
A7	Tx Pair 3 +	O
B7	Tx Pair 3 +	O
D7	Rx Pair 3 +	I
E7	Rx Pair 3 +	I

Pin	Signal	Direction
F16	MSCL (XMC ID ROM clock)	I
F14	MSDA (XMC ID ROM data)	I/O
F10	GA0 (XMC geographic address 0)	I
C12	GA1 (XMC geographic address 1)	I
C14	GA2 (XMC geographic address 2)	I
F2	MRSTI# (XMC reset, active low)	I
C11	BIST	I
A2,A4,A6,A8,A10,A12,A14,A18, B2,B4,B6,B8,B10,B12, B14,B16,B18,C2,C4,C6,C8,C10,C12,C14,C16,C18,D2,D4,D6,D8,D10,D12,D14,D16,D18, E2,E4,E6,E8,E10,E12, E14,E16,E18	Digital Ground	Power
C1,C3,C5,C7	3.3V from host	Power
All others	No Connect	-



CJ2 - Compact PCI and PXI

Connector Types:	2mm hard metric, Framatone HM2R70PA5108N9
Number of Connections:	110, arranged as 5 rows of 22 pins each
Mating Connector:	AMP 352131-1
Baseboard	Duet

CJ2 is the compact PCI connector containing the PXI local buses, triggers and clocks.

CJ2 Pin Number	Function	FPGA Pin Number	Direction (from Duet)
A1	PXI_LBL9	AE19	I/O
A2	PXI_LBR11	AF21	I/O
A3	PXI_LBR7	AJ22	I/O
A4	PCI VIO	-	PWR
A5	CBE5	-	I/O
A6	AD63	-	I/O
A7	AD59	-	I/O
A9	AD56	-	I/O
A10	AD49	-	I/O
A11	AD45	-	I/O
A12	AD42	-	I/O
A13	AD38	-	I/O
A14	AD35	-	I/O
A15	-	-	-
A16	PXI_TRIG1	AG14	I/O
A17	PXI_TRIG2	AJ15	I/O
A18	PXI_TRIG3	AH15	I/O
A19	PXI_LBL2	AK21	I/O

CJ2 Pin Number	Function	FPGA Pin Number	Direction (from Duet)
A20	PXI_LBR4	AF22	I/O
A21	PXI_LBR0	AK24	I/O
A22	-	-	-
B1	GND	-	PWR
B2	PXI_LBR12	AG21	I/O
B3	GND	-	PWR
B4	-	-	-
B5	64EN_N	-	I
B6	AD62	-	I/O
B7	GND	-	PWR
B8	AD55	-	I/O
B9	GND	-	PWR
B10	AD48	-	I/O
B11	GND	-	PWR
B12	AD41	-	I/O
B13	GND	-	PWR
B14	AD34	-	I/O
B15	GND	-	PWR
B16	PXI_TRIG0	AH14	I/O
B17	GND	-	PWR
B18	PXI_TRIG4	AJ16	I/O
B19	GND	AD19	PWR
B20	PXI_LBR5	AG22	I/O
B21	GND	AH22	PWR
B22	-	-	-
C1	PXI_LBL10	AF19	I/O
C2	PCI VIO	-	PWR

CJ2 Pin Number	Function	FPGA Pin Number	Direction (from Duet)
C3	PXI_LBR8	AK22	I/O
C4	CBE7	-	I/O
C5	PCI VIO	-	PWR
C6	AD61	-	I/O
C7	PCI VIO	-	PWR
C8	AD54	-	I/O
C9	PCI VIO	-	PWR
C10	AD47	-	I/O
C11	PCI VIO	-	PWR
C12	AD40	-	I/O
C13	PCI VIO	-	PWR
C14	AD33	-	I/O
C15	-	-	-
C16	-	-	-
C17	-	-	-
C18	PXI_TRIG5	AH16	I/O
C19	PXI_LBL3	AL21	I/O
C20	PXI_LBL0	AH21	I/O
C21	PXI_LBR1	AL24	I/O
C22	-	-	-
D1	PXI_LBL11	AG19	I/O
D2	PXI_LBL7	AJ20	I/O
D3	PXI_LBR9	AL22	I/O
D4	GND	-	PWR
D5	CBE4	-	I/O
D6	GND	-	PWR
D7	AD58	-	I/O
D8	GND	-	PWR

CJ2 Pin Number	Function	FPGA Pin Number	Direction (from Duet)
D9	AD51	-	I/O
D10	GND	-	PWR
D11	AD44	-	I/O
D12	GND	-	PWR
D13	AD37	-	I/O
D14	GND	-	PWR
D15	PXI_LBL6	AE21	I/O
D16	GND	-	PWR
D17	PXI_STAR	AL12	I/O
D18	GND	-	PWR
D19	PXI_LBL4	AE20	I/O
D20	GND	-	PWR
D21	PXI_LBR2	AL23	I/O
D22	-	-	-
E1	PXI_LBL12	AH19	I/O
E2	PXI_LBL8	AD19	I/O
E3	PXI_LBR10	AE21	I/O
E4	CBE6	-	I/O
E5	PAR64	-	I/O
E6	AD60	-	I/O
E7	AD57	-	I/O
E8	AD53	-	I/O
E9	AD50	-	I/O
E10	AD46	-	I/O
E11	AD43	-	I/O
E12	AD39	-	I/O
E13	AD36	-	I/O
E14	AD32	-	I/O

CJ2 Pin Number	Function	FPGA Pin Number	Direction (from Duet)
E15	PXI_LBR6	AH20	I/O
E16	PXI_TRIG7	AJ11	I/O
E17	PXI 10MhZ CLK	AL13	I/O
E18	PXI_TRIG6	AK11	I/O
E19	PXI_LBL5	AF20	I/O
E20	PXI_LBL1	AJ21	I/O
E21	PXI_LBR3	AE22	I/O
E22	-	-	-
F1..F22	DGND	-	POWER

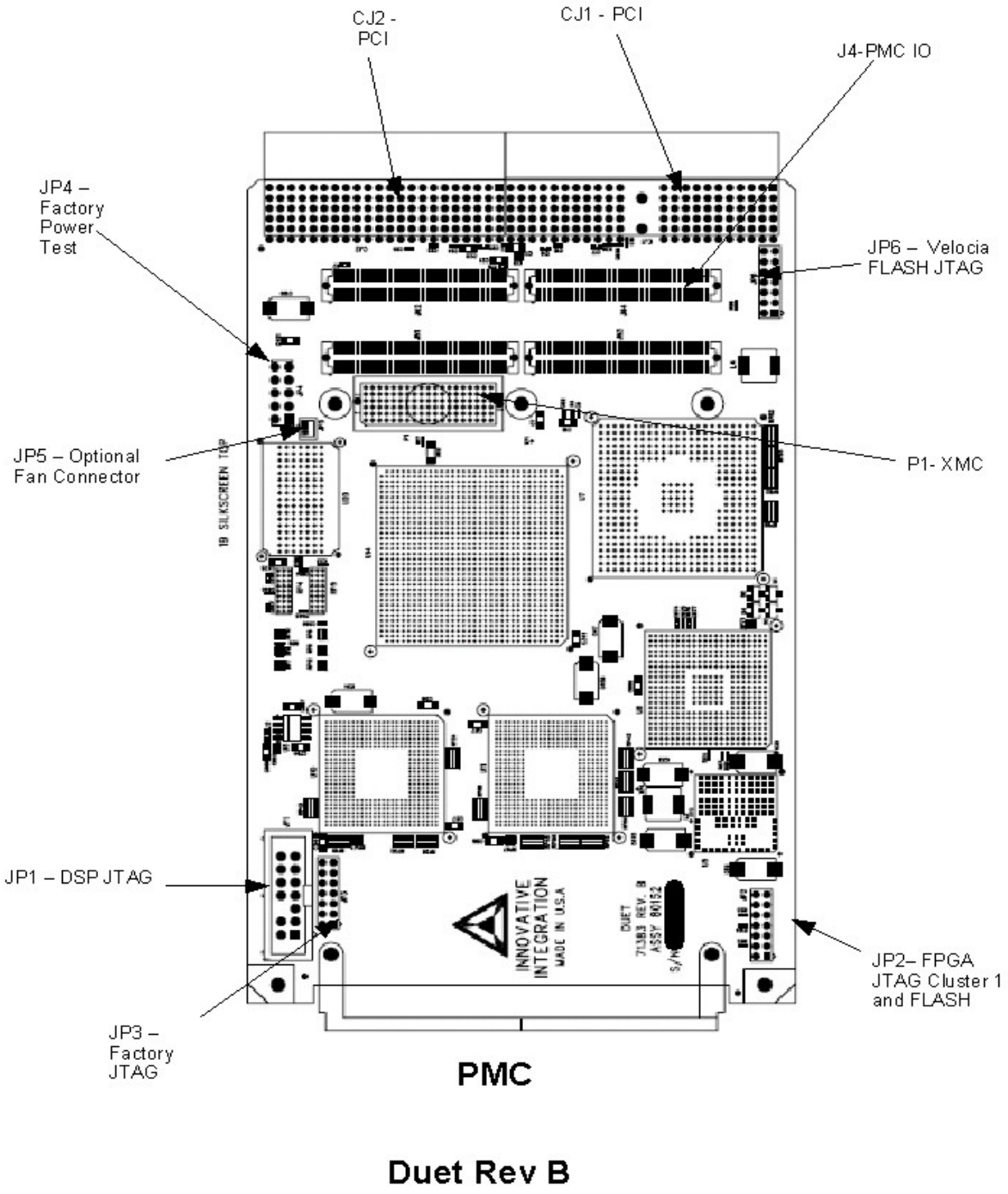


Figure 38. Duet Rev B Board Layout

Chapter 13. *Troubleshooting*

Initialization Problems

The system does not recognize my board(s).

For PCI cards, follow the following steps:

1. Make sure each baseboard is seated in a PCI slot correctly.
2. The device driver must be installed properly. Insure that the proper inf file (see the table for each board's inf file name) is located in the Windows INF folder and iixwdm.sys is located in the Windows System32\Drivers folder.
3. Each baseboard must have an IRQ. Therefore, after booting up, verify that each board does have an IRQ.

- To do this, bring up the Control Panel | System function. Click on the Device Manager tab/button and find the baseboards. Check the Properties | Resources tab for each board.
- If you have conflicting IRQs, you will have to go into your Bios Setup at start-up and change the IRQ of your baseboard(s).

Board	INF File name	Driver
Quadia Baseboard	QuadiaDrvx2K.inf	iixwdm.sys
C64x DSP	C64xDrvx2K.inf	iixwdm.sys
JTAG	JtagDrvx2k.inf, JtagDrvx9x.inf	iixwdm.sys, IIdrvX.vxd

Table 57. Windows driver files

I created an EXE file and when I try to run it, the system requires a DLL which I don't have.

Depending on the settings your application has for building, it may require certain DLLs, such as **borlndmm.dll.** When you try to run your newly created executable, you may get an error such as "Dynamic link library borlndmm.dll can't be found". One cause of this is when you have the project set for dynamic rather than static linking of the Borland VCL packages. While dynamic linking can result in smaller EXEs, dynamic linking can result in dynamic link error messages, such as the one mentioned above, when a DLL is unavailable or not find-able by the application program at invocation.

We recommend static-binding of all executables. To do this:

Bring up the Project Options menu by clicking the Project | Options menu in Builder. Select the **Linker** tab.

Deselect the "Use dynamic RTL" option. This will force the inclusion of the DLLs that are required by your application.

Another possibility is that you have built your application with runtime packages. By building in this fashion, the executable requires certain BPLs to be in the system directory. To build your application without this dependency, click the **Packages** tab on your Project Options menu. Deselect "Build with runtime packages" option.

What DLLs do I have to deploy with my newly created executable?

The following DLLs must reside on the path for any deployed Vista application: Typically, these files are placed in the Windows system directory.

For Windows 9x, the system directory is the **C:\windows\system** directory. For Windows NT, it is **C:\Winnt\system32** directory.

Function	Required DLLs
Intel native signal processing libraries	nsp.dll, nspa6.dll, nspm5.dll, nspm6.dll, nspp6.dll, nsppx.dll, nspw7.dll
Intel native image processing libraries	ipl.dll, iplm5.dll, ipla6.dll, iplm6.dll, iplp5.dll, iplp6.dll, iplpx.dll, iplw7.dll
Innovative device driver DLL	iidrvx40.dll
Innovative Caliente DLL	CalienteDll6.dll (MSVC users runtime only)
Innovative Registration DLL	UserRegister6.dll (BCB users design-time only)

How do I know what DLLs my executable is dependent on?

The Applets\Third Party folder contains an archive call depends20_x86.zip which contains a utility called Depends.exe the provides a utility that allows you to determine the external dependencies of and Windows executable file. Clock File | Open from the menu bar and browse to the name of your application executable. The utility will list all DLLs on which your application is dependent in order to run. Note, however, that Windows programs are always dependent on Windows system DLLs, such as **Advapi32.dll, Kernel32.dll, Version.dll, Comctl32.dll, Gdi32.dll, User32.dll, Ole32.dll and Oleaut32.dll.** These dependencies cannot be eliminated, but will not cause a runtime error, since Windows systems always provide these DLLs.

If the utility exposes DLL dependencies that you would like to remove, follow the steps in the question above "I created an EXE file and when I try to run it, the system requires a DLL which I don't have."

DSP Hardware Problems

The I/O seems like it is not connected or doesn't work.

Double-check the connections to the I/O connector. The most common error is to connect to the wrong pins on the I/O connector.

Check the trigger and timebase setups. For the analog IO on Matador cards, be sure that the timebase you have selected is running, and the trigger method is valid. No data can be collected on these cards without a timebase and an active trigger (start trigger occurred, stop has not yet occurred).

How can I tell what version of logic I am using?

The PCI logic version is reported by the Eeprom applet. When the applet is opened, the current logic version number is displayed.

Quadia Hardware Problems

How do I update the logic?

The logic may be updated using the Eeprom applet. This applet updates only the PCI logic on the baseboard. The update process is straightforward and is generally trouble-free. Two big mistakes that can occur are putting the wrong logic into the card and terminating the application before completion. If you put the wrong logic into the card, this may be recoverable by just repeating the programming process before power-cycling the PC, provided that the logic file is an earlier Quadia-compatible version. If the file is completely wrong, **DO NOT TURN THE PC OFF.** So long as power is applied, the logic image in the FLASH is not yet used and you can still reprogram the correct one into the card.

If the Eeprom program consistently crashes or terminates early for any reason, the card must be returned to Innovative for reprogramming. Sorry.

I updated the logic, but it did not work.

The most common reasons for this are that the wrong logic image was used, or the card was not power-cycled between tests. The logic is not reloaded until the card is powered up again.